sgcWebSockets 2026.10

October 2026

Documentation for CBuilder

Copyright © 2012-2026 eSeGeCe Software
info@esegece.com
www.esegece.com

Contents

Introduction **31**

Overview **36**

Editions 36

Installation 38

Install **41**

Install Setup 41

Install Package 52

Install Errors 59

Configure Install 63

Install sgcIndy Package 65

Configure ZLib 71

QuickStart **72**

Overview 72

QuickStart WebSockets 74

QuickStart HTTP 76

Threading Flow 78

Build 80

Build OSX Application 81

Build Android Application 83

Build iOS Application 84

Fast Performance Server 86

Memory Manager 89

SSL 92

SChannel 93

Native Android TLS 96

Native Apple TLS 97
Running without OpenSSL 98
OpenSSL 100
OpenSSL on Windows | sgcWebSockets 103
OpenSSL OSX 105
OpenSSL Android 107
OpenSSL iOS 108
OpenSSL Own CA Certificates 110
OpenSSL PKCS#12 certificates | sgcWebSockets 112
OpenSSL Verify Certificate 113
OpenSSL Load Additional Functions 114
OpenSSL Static Linking | sgcWebSockets 116
Indy 117

Topics 119

WebSocket Events 119
WebSocket Parameters Connection 120
Using inside a DLL 121
Windows Service 122
Web Browser Test 123
Custom Sub-Protocols 124
Authentication 125
Secure Connections 127
HeartBeat 129
WatchDog 130
Logs 131
HTTP 132
Broadcast and Channels 133
Bindings 134
Post Big Files 135
Compression 137

HTTP Compression 138
Custom Objects 139
Groups 140
IOCP 142
EPOLL 144
TLS Provider 146
Native TLS 149
ALPN 154
Forward HTTP Requests 155
Quality Of Service 156
Queues 158
Transactions 160
Message History and Reconnect Recovery 162
TCP Connections 164
SubProtocol 165
Throttle 166
Server-Sent Events (SSE) | sgcWebSockets 167
LoadBalancing 169
Files 170
Proxy 171
Fragmented Messages 172

Components 173

TsgcWebSocketClient 173
Connect WebSocket Server 175
Client Open Connection 176
Client Close Connection 178
Client Keep Connection Open 179
Dropped Disconnections 180
Connect TCP Server 181
Connections TIME_WAIT 182

WebSocket Redirections 183
Connect Secure Server 184
Certificates OpenSSL 185
Certificates SChannel 186
SChannel Certificate Revocation 188
SChannel Get Connection Info 190
Client Send Text Message 191
Client Send Binary Message 192
Client Send Text and Binary Message 193
Receive Text Messages 194
Receive Binary Messages 195
Client Authentication 196
Client Exceptions 199
Client WebSocket HandShake 200
Client Register Protocol 201
Client Proxies 202
TsgcTCPClient 203
TsgcTCPClient 204
TsgcWebSocketServer 205
Server Start 207
Server Bindings 208
Server Startup Shutdown 209
Server Keep Active 210
Server SSL 211
TLS and plain HTTP on the same port 213
Server SSL SChannel 215
Server SSL SChannel Client Certificate Revocation 218
Server Verify Certificate 220
Server Keep Connections Alive 221
Server Plain TCP 222
Server Close Connection 223

Client Connections 224
Server Authentication 225
Server Send Text Message 227
Server Send Binary Message 228
Server Receive Text Message 229
Server Receive Binary Message 230
Server Read Headers from Client 231
TsgcWebSocketHTTPServer 232
HTTP Server Requests 234
HTTP Dispatch Files 235
HTTP/2 Server 236
HTTP/2 Server Push 237
HTTP/2 Alternate Service 239
HTTP/2 Server Threads 240
HTTP 404 Error without Response Body 242
HTTP Server Sessions 243
HTTP Server Stream Video 246
Server SSL SChannel 247
TsgcHTTPServer 250
HTTP Server Requests 252
HTTP Dispatch Files 253
HTTP/2 Server 254
HTTP/2 Server Push 255
HTTP/2 Alternate Service 257
HTTP/2 Server Threads 258
HTTP 404 Error without Response Body 260
HTTP Server Sessions 261
HTTP Server Stream Video 264
Server SSL SChannel 265
TsgcHTTPRESTServer 268
TsgcHTTPServer_Users 269

TsgcHTTPServer_Tenancy 270
TsgcHTTPServerStats 272
TsgcWebSocketServer_HTTPAPI 274
HTTP.sys URL reservation | sgcWebSockets 276
HTTPAPI FineTune 278
HTTPAPI Server SSL 280
Self-Signed Certificates 281
Disable HTTP/2 in the HTTP API server | sgcWebSockets 282
HTTPAPI Custom Headers 283
HTTPAPI Send Text Response 284
HTTPAPI Send File Response 285
HTTPAPI OnDisconnect not fired 287
TsgcWebSocketClient_WinHTTP 288
TsgcWebSocketFirewall 289
Firewall: Blacklist and Whitelist 296
Firewall: Brute Force Protection 298
Firewall: SQL Injection and XSS Detection 300
Firewall: Rate Limiting and Flood Protection 303
Firewall: Bot Detection 306
Firewall: Real Client IP Behind a Reverse Proxy 309
TsgcWSRateLimiter 313
TsgcWSCircuitBreaker 325
TsgcWSAPIKeyManager 336
TsgcWSCluster 348
Cluster: Mesh Engine 352
Cluster: Redis Engine 354
Cluster: Cluster-Wide Presence 356
TsgcWebSocketLoadBalancerServer 357
TsgcWebSocketProxyServer 358
TsgcIWWebSocketClient 359
TsgcWSConnection 361

Protocols 363
Protocols Javascript 365
Protocol MQTT 368
TsgcWSPClient_MQTT 370
Client MQTT Connect 371
Connect Mosquitto MQTT Servers 372
Client MQTT Sessions 373
Client MQTT Version 374
MQTT Publish Subscribe 375
MQTT Topics 376
MQTT Subscribe 377
MQTT Publish Message 378
MQTT Receive Messages 379
MQTT Publish and Wait Response 380
MQTT Clear Retained Messages 381
Protocol AMQP 382
TsgcWSPClient_AMQP 383
Client AMQP Connect 386
Client AMQP Disconnect 387
AMQP Channels 388
AMQP Exchanges 390
AMQP Queues 392
AMQP Publish Messages 395
AMQP Consume Messages 396
AMQP Get Messages 398
AMQP QoS 399
AMQP Transactions 400
Protocol Kafka 402
TsgcWSPClient_Kafka 403
Client Kafka Connect 405
Client Kafka Disconnect 406

Kafka Produce Messages 407
Kafka Consume Messages 408
Kafka Consumer Groups 409
Kafka Topics 410
Kafka Offsets 411
Protocol AMQP1 412
TsgcWSPClient_AMQP1 414
Client AMQP1 Connect 417
Client AMQP1 Disconnect 418
Client AMQP1 Idle Timeout Connection 419
Client AMQP1 Connection State 420
Client AMQP1 Authentication 421
Client AMQP1 Azure MessageBus 422
AMQP1 Sessions 424
AMQP1 Links 426
AMQP1 Sender Links 427
AMQP1 Receiver Links 430
AMQP1 Send Message 432
AMQP1 Read Message 434
Protocol STOMP 435
TsgcWSPClient_STOMP 436
TsgcWSPClient_STOMP_RabbitMQ 437
TsgcWSPClient_STOMP_ActiveMQ 438
Protocol WebRTC 439
TsgcWSPServer_WebRTC 440
Protocol WebRTC Javascript 441
Protocol WAMP 442
TsgcWSPServer_WAMP 443
TsgcWSPClient_WAMP 444
Protocol WAMP Javascript 445
Subscribers 448

Publishers 449
Simple RPC 450
RPC Progress Results 451
Protocol WAMP2 453
TsgcWSPClient_WAMP2 454
Protocol Default 457
TsgcWSPServer_sgc 459
TsgcWSPClient_sgc 460
TsgcIWWSPClient_sgc 461
Protocol Default Javascript 462
Protocol Dataset 466
TsgcWSPServer_Dataset 467
TsgcWSPClient_Dataset 468
TsgcIWWSPClient_Dataset 469
Protocol Dataset Javascript 470
Protocol Dataset Replicate Table 473
Protocol Dataset Notify Updates 474
Protocol Files 475
TsgcWSPServer_Files 476
TsgcWSPClient_Files 477
TsgcWSMessageFile 478
How Send Files To Server 479
How Send Files To Clients 480
How Send Big Files 481
Protocol Presence 482
TsgcWSPServer_Presence 483
TsgcWSPresenceMessage 484
TsgcWSPClient_Presence 485
Protocol Presence Javascript 486
Protocol E2EE 489
TsgcWSPServer_E2EE 491

TsgcWSPClient_E2EE 492
APIs 493
Binance API | sgcWebSockets 496
Binance Connect WebSocket API 504
Binance Subscribe WebSocket Channel 505
Binance Get Market Data 506
Binance Private REST API 507
Binance Trade Spot 508
Binance Private Requests Time 510
Binance Withdraw 511
Binance Futures API | sgcWebSockets 512
API Binance Futures Trade 519
Socket.IO Client | sgcWebSockets 520
API Coinbase 522
Coinbase Connect WebSocket API 527
Coinbase Subscribe WebSocket Channel 528
Coinbase Get Market Data 529
Coinbase Private REST API 530
Coinbase Private Requests Time 531
Coinbase Place Orders 532
Coinbase SandBox Account 533
API SignalRCore 534
API SignalR 540
API Kraken 543
API Kraken WebSockets Public 545
API Kraken WebSockets Private 551
API Kraken REST Public 554
API Kraken REST Private 556
API Kraken v2 559
API Kraken v2 WebSockets Public 562
API Kraken v2 WebSockets Private 567

API Kraken Futures 573
API Kraken Futures WebSockets Public 575
API Kraken Futures WebSockets Private 582
API Kraken Futures REST Public 588
API Kraken Futures REST Private 590
API Pusher 595
API Bitmex 603
Bitmex Connect WebSocket API 607
Bitmex Subscribe WebSocket Channel 608
How to Place a Bitmex Order 609
API Bitfinex 611
API Kucoin 615
Kucoin Connect WebSocket API 622
Kucoin Subscribe WebSocket Channel 623
Kucoin Get Market Data 624
Kucoin Private REST API 625
Kucoin Trade Spot 626
Kucoin Private Requests Time 628
KuCoin Futures API | sgcWebSockets 629
Kucoin Futures Connect WebSocket API 634
Kucoin Futures Subscribe WebSocket Channel 635
Kucoin Futures Get Market Data 636
Kucoin Futures Private REST API 637
Kucoin Futures Trade 638
Kucoin Futures Private Requests Time 641
API 3Commas 642
OKX API | sgcWebSockets 647
XTB API | sgcWebSockets 654
Bybit API | sgcWebSockets 657
API Forex 662
Forex Connect REST API 664

Forex Trade Orders 665
Forex Account Queries 667
Forex Market Data 668
Forex Connect WebSocket API 669
Forex Subscribe Streams 670
API Blockchain 672
API Cex 674
API Cex Plus 682
API Discord 687
API OpenAI 690
MEXC API | sgcWebSockets 692
MEXC Futures API | sgcWebSockets 696
API Bitget 700
Gate.io API | sgcWebSockets 703
API Deribit 705
API Crypto.com 708
HTX API | sgcWebSockets 710
API Whatsapp 712
WhatsApp Create App 716
WhatsApp Phone Number Id 718
WhatsApp Token 719
WhatsApp Webhook 720
WhatsApp Security 721
WhatsApp Send Messages 722
WhatsApp Send Interactive Messages 725
WhatsApp Send Template Messages 729
WhatsApp Receive Messages and Status Notifications 731
WhatsApp Send Files 733
WhatsApp Download Media 735
Telegram API (TDLib) | sgcWebSockets 736
Send Telegram Message With Inline Buttons 744

Send Telegram Message With Buttons 745
Send Telegram Message Bold 746
Telegram Chat not found as Bot 747
Telegram Sponsored Messages 748
Send Telegram Invoice Message 749
Telegram Get SuperGroup Members 750
Add Telegram Proxy 751
Register Telegram User 752
RCON 753
CryptoHopper 754
WebPush 759
TsgcWSAPIServer_WebPush 760
TsgcWebPush_Client 761
Extensions 762
WebSocket permessage-deflate compression | sgcWebSockets 763
Deflate-Frame 764
MCP 765
TsgcWSAPIServer_MCP 767
MCP Server Sessions 770
MCP Server Protocol Versions 771
MCP Server Subscriptions 773
MCP Server Tasks 774
MCP Server Logging and Progress 776
MCP Server Tools 777
MCP Server Prompts 781
MCP Server Resources 784
MCP Server Roots 787
MCP Server Sampling 789
MCP Server Elicitation 791
MCP Server Multi Round-Trip Requests 793
TsgcWSAPIClient_MCP 795

MCP Client Protocol Versions 796
MCP Client Subscriptions 798
MCP Client Tasks 799
MCP Client Tools 801
MCP Client Prompts 803
MCP Client Resources 805
MCP Client Roots 807
MCP Client Sampling 809
MCP Client Elicitation 811
MCP Client Multi Round-Trip Requests 813
MCP Client OAuth2 Authentication 815
MCP Client Authorization 817
OpenAI 819
TsgcHTTP_API_OpenAI 826
OpenAI Completion 827
OpenAI Chat 828
OpenAI Edit 829
OpenAI Audio 830
OpenAI Moderation 831
OpenAI RealTime 832
OpenAI Responses 833
OpenAI Speech 834
OpenAI Fine-Tuning 835
OpenAI Batch 836
OpenAI Uploads 837
OpenAI Applications 838
TsgcAIOpenAIAssistant 839
TsgcAIOpenAIAssistant File Search 841
TsgcAIOpenAIAssistant Streaming 843
TsgcAIOpenAIAssistant Function Calling 845
OpenAI Audio 847

TsgcAudioRecorderMCI 848
TsgcAudioPlayerMCI 849
TsgcTextToSpeechSystem 850
TsgcTextToSpeechGoogle 851
TsgcTextToSpeechAmazon 852
TsgcAIChat 853
TsgcAIChat Structured Output 854
TsgcAIChat Documents and PDF 857
TsgcAIDocumentExtractor 859
TsgcAIOCRTesseract 861
TsgcAIOCRWindows 863
Extract data from an invoice 865
TsgcAIVoiceAgent 868
Voice assistant with tools 871
Barge-in and turn detection 873
Telephony with G.711 875
TsgcAIOpenAIChatBot 877
TsgcAIOpenAITranslator 878
TsgcAIOpenAIEmbeddings 879
TsgcAIDatabaseVectorFile 880
TsgcAIDatabaseVectorPinecone 881
Embeddings Create Vectors 882
Embeddings ChatBot 883
Pinecone 884
Anthropic 887
TsgcHTTP_API_Anthropic 891
Anthropic Messages 892
Anthropic Vision 894
Anthropic Tool Use 896
Anthropic Models 898
Anthropic Batches 899

Anthropic Extended Thinking 900
Anthropic Documents 902
Anthropic Prompt Caching 904
Anthropic Citations 906
Anthropic Web Search 907
Anthropic Structured Outputs 909
Anthropic Files 911
Anthropic MCP Connector 913
Gemini 915
TsgcHTTP_API_Gemini 917
DeepSeek 918
TsgcHTTP_API_DeepSeek 919
DeepSeek Messages 920
DeepSeek Vision 921
DeepSeek Models 922
Gemini 923
Gemini Messages 925
Gemini Vision 927
Gemini Models 928
Gemini Structured Outputs 929
Gemini Token Counting 930
Gemini Embeddings 931
Gemini Tool Use 932
Ollama 934
TsgcHTTP_API_Ollama 936
Ollama Messages 937
Ollama Models 938
Ollama Embeddings 939
Grok 940
TsgcHTTP_API_Grok 941
Grok Messages 942

Grok Vision 943
Grok Models 944
Mistral AI 945
TsgcHTTP_API_Mistral 947
Mistral Messages 948
Mistral Vision 949
Mistral Models 950
Mistral Embeddings 951
Cryptography 952
Hashing & Message Authentication 955
Password Hashing & Key Derivation 956
Symmetric Encryption 957
Public Key Cryptography 959
Post-Quantum Cryptography 961
Certificates & PKI 964
One Time Passwords 966
Fuzzy Hashing (TLSH) 967
WinZip AE-2 Archive Encryption 968
FIPS Mode 969
IoT 972
AWS IoT Core MQTT client | sgcWebSockets 973
IoT Azure MQTT Client 981
HTTP 986
HTTP2 987
TsgcHTTP2Client 988
Request HTTP/2 Method 990
HTTP/2 Server Push 991
HTTP/2 Download File 992
HTTP/2 Partial Responses 993
HTTP/2 Headers 994
Client Close Connection 995

Client Keep Connection Active ... 996
HTTP/2 Reason Disconnection ... 997
Client Pending Requests ... 998
Client Authentication ... 999
HTTP/2 and OAuth2 ... 1000
TsgcHTTP2ConnectionClient ... 1001
TsgcHTTP2RequestProperty ... 1002
TsgcHTTP2ResponseProperty ... 1003
Apple Push Notifications (APNs) | sgcWebSockets ... 1004
Generate a Remote Notification APNs ... 1005
Sending Notification Requests to APNs ... 1006
Token-Based Connection to APNs ... 1007
Certificate-Based Connection to APNs ... 1008
QUIC ... 1010
TsgcQUICClient ... 1014
TsgcQUICServer ... 1018
HTTP/3 ... 1022
TsgcHTTP3Client ... 1026
TsgcHTTP3Server ... 1031
WebTransport ... 1035
TsgcWebTransportSession ... 1039
TsgcWebTransportStream ... 1043
gRPC ... 1047
sgcProtoBuf ... 1048
TsgcGRPCClient ... 1050
Google Cloud gRPC Services ... 1054
Google Cloud Pub/Sub ... 1055
Google Cloud Speech-to-Text ... 1057
Google Cloud Translation ... 1059
Google Cloud Vision ... 1061
Google Cloud Natural Language ... 1063

Google Cloud Storage 1065
Google Cloud BigQuery Storage 1067
Google Vertex AI 1069
HTTP1 1071
Server-Sent Events 1076
TsgcSSEClient 1077
Resume with Last-Event-ID 1079
TsgcSSEEvent 1080
TsgcSSEParser 1081
OAuth2 1082
TsgcHTTP_OAuth2_Client 1083
OAuth2 Client for Web Applications 1087
OAuth2 Client for Desktop Applications 1088
TsgcHTTP_OAuth2_Client_Google 1089
TsgcHTTP_OAuth2_Client_Microsoft 1090
Authorization Code Grant (RFC 6749) 1091
Authorization Code with PKCE (RFC 7636) 1093
Client Credentials Grant (RFC 6749) 1095
Resource Owner Password Credentials Grant (RFC 6749) 1097
Device Authorization Grant (RFC 8628) 1099
DPoP - Demonstrating Proof of Possession (RFC 9449) 1101
Identity Assertion Authorization Grant 1105
TsgcHTTP_OAuth2_Server 1108
OAuth2 Server Example 1111
OAuth2 Customize Sign-In HTML 1115
OAuth2 Server Endpoints 1116
OAuth2 Register Apps 1117
OAuth2 Recover Access Tokens 1118
OAuth2 Server Authentication 1119
OAuth2 None Authenticate URLs 1120
Authorization Code Grant 1121

Client Credentials Grant 1123
Resource Owner Password Credentials Grant 1125
Refresh Token Grant 1127
Device Authorization Grant (RFC 8628) 1129
Token Revocation (RFC 7009) 1131
Token Introspection (RFC 7662) 1133
DPoP Validation (RFC 9449) 1135
TsgcHTTP_OAuth2_Server_Provider 1137
OAuth2 Provider Azure AD 1139
OAuth2 Provider Private Endpoints 1140
OAuth2 Provider Authentication 1141
OAuth2 Provider Requests 1143
JWT 1144
TsgcHTTP_JWT_Client 1146
TsgcHTTP_JWT_Server 1149
WebAuthn 1150
Passkeys 1151
TsgcWSAPIServer_WebAuthn 1154
WebAuthn Registration 1158
WebAuthn Registration Request 1161
WebAuthn Registration Response 1162
WebAuthn Registration Result 1164
WebAuthn Authentication 1165
WebAuthn Authentication Request 1167
WebAuthn Authentication Response 1168
WebAuthn Authentication Result 1169
WebAuthn MDS 1170
WebAuthn Authorization 1171
WebAuthn Authorization HTTP 1172
WebAuthn Authorization WebSocket 1173
Webauthn Javascript Client 1174

TsgcTOTPAuthenticator 1178
TsgcLDAPClient 1180
TsgcSAMLServiceProvider 1182
TsgcSAMLResult 1185
TsgcHTTP_OIDC_Client 1186
TsgcOIDCJWKS 1189
TsgcMailOAuth2 1191
Amazon SQS 1193
Google OAuth2 Keys 1198
Google Service Accounts 1204
Google Cloud Pub/Sub 1208
Google Calendar 1217
Google Calendar Sync Calendars 1223
Google Calendar Sync Events 1224
Google Calendar RefreshToken 1225
Google Calendar Service Account 1226
Google Cloud FCM 1227
TsgcWebView2 1230
WebView2 Navigation 1232
WebView2 JavaScript 1234
WebView2 Cookies 1236
WebView2 Downloads 1238
WebView2 Settings 1240
WebView2 Advanced Features 1242
TsgcUDPCLient 1245
TsgcUDPServer 1246
STUN 1247
TsgcSTUNClient 1248
STUN Client UDP Retransmissions 1250
STUN Client Long Term Credentials 1251
STUN Client Attributes 1252

TsgcSTUNServer 1253
STUN Server Long Term Credentials 1254
STUN Server Alternate Server 1255
TURN 1256
TsgcTURNClient 1257
TURN Client Allocate IP Address 1259
TURN Client Create Permissions 1260
TURN Client Send Indication 1261
TURN Client Channels 1262
TsgcTURNServer 1263
TURN Server Long Term Credentials 1265
TURN Server Allocations 1266
ICE 1267
TsgcICEClient 1268
ICE Gather Candidates 1269
ICE Pair Candidates 1270
TsgcRTCPeerConnection 1271
RTCPeerConnection WebSocket Server 1273
RTCPeerConnection WebSocket Client 1274
RTCPeerConnection STUN TURN 1275
RTCPeerConnection Signaling 1276
RTCPeerConnection ICE 1277
RTCPeerConnection DTLS 1278
RTCPeerConnection Data 1279
Datasnap 1280
TsgcWSHTTPWebBrokerBridgeServer 1281
TsgcWSHTTP2WebBrokerBridgeServer 1283
TsgcWSServer_HTTPAPI_WebBrokerBridge 1284
sgcHTML pages on DataSnap Servers 1285

OpenAPI 1287

OpenAPI 1287
OpenAPI Parser Pascal 1289
OpenAPI Additional Properties 1295
OpenAPI Command Line 1297
OpenAPI Library 1300
OpenAPI API 1302
OpenAPI Client 1303
OpenAPI Server 1307
OpenAPI Server Spec-First 1310
OpenAPI Server Code-First 1312
OpenAPI Amazon AWS 1317
OpenAPI Amazon AWS Credentials 1323
OpenAPI Amazon AWS S3 1325
OpenAPI Google Cloud 1326
OpenAPI Google Cloud OAuth2 1331
OpenAPI Google Cloud Service Accounts 1334
OpenAPI Google Cloud PubSub 1339
OpenAPI Google Cloud Calendar 1340
OpenAPI Google Drive 1341
OpenAPI Microsoft 1343
OpenAPI Microsoft Tenant 1348
OpenAPI Microsoft Register Application 1349
OpenAPI Microsoft OAuth2 Code 1352
OpenAPI Microsoft OAuth2 Credentials 1354
OpenAPI Microsoft Graph 1356
APIs 1357
AbstractApi Geolocation 1358

HTML Components 1359

HTML Components 1359
QuickStart 1361

Runtime vs Design-Time 1363
Designing in the Browser 1365
AI Page Generator 1367
MCP Apps 1371
CRUD Pages 1374
TsgcHTMLComponent — Common Properties 1377
Node / Composition Layer 1379
Localization 1382
Routing 1385
Sessions, CSRF and Authentication 1390
Testing sgcHTML Pages 1398
Navigation and Live Updates 1402
WebSocket or SSE 1406
AI Data Pack 1410
Web Push 1415
ASP.NET Core Razor (Tag Helpers) 1418
Razor Grid 1422
Razor Forms and Bound Inputs 1427
Razor AutoComplete 1430
Razor Dialog 1432
Razor Chart 1434
Integrating Third-Party JavaScript Libraries 1437
TsgcHTMLComponent_Chart 1440
TsgcHTMLComponent_Gauge 1445
Instruments 1450
TsgcHTMLComponent_LinearGauge 1453
TsgcHTMLComponent_LED 1456
TsgcHTMLComponent_LEDBar 1459
TsgcHTMLComponent_SegmentDisplay 1462
TsgcHTMLComponent_Annunciator 1465
TsgcHTMLAnnunciatorTile 1468

TsgcHTMLComponent_Odometer 1469
TsgcHTMLComponent_VUMeter 1471
TsgcHTMLComponent_AnalogClock 1473
TsgcHTMLComponent_Compass 1476
TsgcHTMLComponent_Knob 1479
TsgcHTMLComponent_ToggleSwitch 1482
TsgcHTMLComponent_PushButton 1485
TsgcHTMLComponent_NumericStepper 1488
TsgcHTMLComponent_StripChart 1491
TsgcHTMLComponent_Oscilloscope 1494
TsgcHTMLComponent_SCADAPanel 1497
TsgcHTMLInstrumentsBinder 1502
TsgcHTMLInstrumentsMQTTBinder 1505
TsgcHTMLComponent_Diagram 1510
TsgcHTMLComponent_Map 1512
TsgcHTMLComponent_Sparkline 1514
TsgcHTMLComponent_Heatmap 1517
TsgcHTMLComponent_TreeMap 1521
TsgcHTMLComponent_CandlestickChart 1525
TsgcHTMLComponent_QRCode 1530
TsgcHTMLComponent_Barcode 1533
TsgcHTMLComponent_NavBar 1536
TsgcHTMLComponent_Sidebar 1539
TsgcHTMLComponent_Site 1542
TsgcHTMLComponent_Breadcrumb 1548
TsgcHTMLComponent_Tabs 1550
TsgcHTMLComponent_Pagination 1553
TsgcHTMLComponent_Toolbar 1556
TsgcHTMLComponent_TreeView 1558
TsgcHTMLComponent_Stepper 1560
TsgcHTMLComponent_Dropdown 1563

TsgcHTMLComponent_ButtonGroup 1566
TsgcHTMLComponent_ListGroup 1569
TsgcHTMLDashboardLayout 1572
TsgcHTMLComponent_Grid 1575
TsgcHTMLComponent_DataTable 1586
TsgcHTMLComponent_Calendar 1590
TsgcHTMLComponent_Scheduler 1593
TsgcHTMLComponent_Timeline 1595
TsgcHTMLComponent_KanbanBoard 1598
TsgcHTMLComponent_Gantt 1603
TsgcHTMLComponent_PivotTable 1608
TsgcHTMLComponent_TreeGrid 1617
TsgcHTMLComponent_AuditTrail 1621
TsgcHTMLComponent_Form 1625
TsgcHTMLComponent_Edit 1629
TsgcHTMLComponent_Memo 1632
TsgcHTMLComponent_CheckBox 1635
TsgcHTMLComponent_RadioGroup 1637
TsgcHTMLComponent_Select 1640
TsgcHTMLComponent_InputGroup 1643
TsgcHTMLComponent_AutoComplete 1645
TsgcHTMLComponent_DatePicker 1649
TsgcHTMLComponent_FileUpload 1652
TsgcHTMLComponent_RichEditor 1655
TsgcHTMLComponent_Rating 1658
TsgcHTMLComponent_MultiSelect 1661
TsgcHTMLComponent_Transfer 1664
TsgcHTMLComponent_TimePicker 1668
TsgcHTMLComponent_DateTimePicker 1671
TsgcHTMLComponent_DateRangePicker 1674
TsgcHTMLComponent_Slider 1677

TsgcHTMLComponent_RangeSlider 1681
TsgcHTMLComponent_ColorPicker 1684
TsgcHTMLComponent_SignaturePad 1687
TsgcHTMLComponent_NumPad 1691
TsgcHTMLComponent_CameraScanner 1696
TsgcHTMLComponent_Geolocation 1701
TsgcHTMLComponent_Panel 1706
TsgcHTMLComponent_StatCard 1709
TsgcHTMLComponent_Accordion 1712
TsgcHTMLComponent_Carousel 1714
TsgcHTMLComponent_Image 1717
TsgcHTMLComponent_Avatar 1719
TsgcHTMLComponent_Video 1722
TsgcHTMLComponent_Splitter 1725
TsgcHTMLComponent_PDFViewer 1728
TsgcHTMLComponent_Modal 1733
TsgcHTMLComponent_Offcanvas 1736
TsgcHTMLComponent_Popover 1738
TsgcHTMLComponent_Toast 1741
TsgcHTMLComponent_Snackbar 1743
TsgcHTMLComponent_Notification 1746
TsgcHTMLComponent_NotificationInbox 1750
TsgcHTMLComponent_NotificationPreferences 1751
TsgcHTMLWebPush 1752
TsgcHTMLComponent_Spinner 1754
TsgcHTMLComponent_Placeholder 1757
TsgcHTMLComponent_EmptyState 1760
TsgcHTMLComponent_ProgressBar 1764
TsgcHTMLComponent_Badge 1767
TsgcHTMLComponent_Chip 1770
TsgcHTMLComponent_ContextMenu 1773

TsgcHTMLComponent_CommandPalette 1776
TsgcHTMLComponent_Presence 1781
TsgcHTMLComponent_ActivityFeed 1785
TsgcHTMLComponent_JobProgress 1789
TsgcHTMLComponent_LogViewer 1794
TsgcHTMLComponent_ChatBox 1799
TsgcHTMLComponent_Chat 1802
TsgcHTMLComponent_AIChat 1805
TsgcHTMLAIAssistant 1808
TsgcHTMLComponent_InlineAIPrompt 1811
TsgcHTMLAIColumns 1814
TsgcHTMLGridOps 1816
TsgcHTMLPivotOps 1818
TsgcHTMLAuth 1819
TsgcHTMLSessionStore_Memory 1824
TsgcHTMLSessionStore_File 1827
TsgcHTMLComponent_Login 1829
TsgcHTMLComponent_SocialLogin 1833
TsgcHTMLComponent_OAuthCallback 1836
TsgcHTMLComponent_WebAuthnLogin 1839
TsgcHTMLComponent_UserManagement 1842
TsgcHTMLComponent_ImpersonateBanner 1847
TsgcHTMLComponent_RolesPermissions 1850
TsgcHTMLEngine_Server 1855
TsgcHTMX_Engine_Server 1857
TsgcHTMX_Engine_Server_WebBrokerBridge 1860
TsgcHTMX_Engine_Server_WebBroker 1863
TsgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge 1866
TsgcHTMLDesignServer 1869
TsgcHTMLPageBuilder 1873
TsgcHTMLTemplate_Bootstrap 1875

TsgcHTMLTemplate_HTMX 1879
TsgcHTMLThemeController 1881
TsgcHTMLThemeBuilder 1883
TsgcHTMX_Fragment 1885
TsgcHTMX_Router 1887

Demos 1889

Server Chat 1889
Client Chat 1891
Client 1892
Client MQTT 1893
Client SocketIO 1895
Server Monitor 1896
Server Snapshots 1899
Client Snapshots 1900
Upload File 1901
Server Authentication 1903
KendoUI_Grid 1904
ServerSentEvents 1906
Server WebRTC 1907
Telegram Client 1908

Third-parties 1910

Coturn 1910

License 1912

License 1912

Index 1914

PDF-Back-Cover 1929

Introduction

sgcWebSockets is a professional-grade component library for real-time communication, messaging, and AI integration. It provides production-ready implementations of WebSockets, HTTP/2, MQTT, AMQP, WebRTC, Server-Sent Events, and over 30 third-party API connectors, all accessible through a consistent, event-driven component architecture. Whether you are building trading platforms, IoT gateways, AI-powered applications, or enterprise messaging systems, sgcWebSockets delivers the networking foundation you need.
The library supports **Delphi 7 through Delphi 13**, **C++Builder**, **Lazarus/FreePascal**, and **.NET** (Framework 2.0+, .NET Core 1.0+, .NET 5–9, .NET Standard). Applications can target Windows, macOS, Linux, iOS, and Android from a single codebase using VCL and FireMonkey.

Delphi / C++Builder / FreePascal Features

Core WebSocket

- Fully functional **multithreaded WebSocket server** implementing **RFC 6455**.
- **Text** and **Binary** message support.
- **Message Compression** using PerMessage_Deflate extension (**RFC 7692**).
- Built-in **WatchDog** and **HeartBeat** for automatic reconnection and connection monitoring.
- **Load Balancing** Server for distributing connections across multiple backends.
- **Proxy Server** component allowing web browsers to connect to any TCP server.
- Client connections through **HTTP Proxy** and **SOCKS Proxy** servers.
- Events: OnConnect, OnDisconnect, OnMessage, OnError, OnHandshake.

High-Performance Servers

- **Microsoft HTTP Server API** (HTTP.SYS) with **IOCP** for high-performance Windows servers.
- Indy servers with **IOCP** (Windows), **EPOLL** (Linux), or default one-thread-per-connection model.
- **WebSocket** and **HTTP/2** connections through the **same port**.
- **WebBroker Server** supporting **DataSnap**, **HTTP/2**, and **WebSocket** on a single port.
- Client WebSocket based on **WinHTTP API**.

Security and Authentication

- **SSL/TLS** for server and client components. **OpenSSL 1.1.1** and **3.0** supported. Client supports **SChannel** on Windows.
- **OAuth 2.0** and **JWT** (JSON Web Token) client and server components.
- **WebAuthn Server Protocol** for passkey-based authentication.
- **TOTP** two-factor authentication, **LDAP** and Active Directory client, and **OAuth 2.0 for mail** (SMTP, IMAP and POP3 with Microsoft 365 and Gmail), in sgcWebSockets Enterprise and the sgcAuth pack.
- **SAML 2.0** single sign-on service provider and **OpenID Connect** client with ID token validation, in sgcWebSockets Enterprise and the sgcAuth pack.
- **E2EE** (End-To-End Encryption) sub-protocol.

AI and Machine Learning

- **OpenAI** and **Claude** integration with **MCP Server** and **MCP Client** components.
- **ChatBot** and **Translator** application components with voice command support.
- **Embeddings** and **Vector Database** support (file-based and Pinecone).
- **Text-To-Speech** via system default, Google Cloud, and Amazon AWS.

HTTP/2 Protocol

- Full **HTTP/2 client and server** implementation.
- **Server-Sent Events** (push notifications) over HTTP.

gRPC

- **gRPC client** over HTTP/2 with the four RPC patterns (unary, server streaming, client streaming and bidirectional streaming), retries, client-side load balancing, health checking and server reflection.
- Typed **Google Cloud** gRPC interfaces: Pub/Sub, Speech-to-Text, Translation, Vision, Natural Language, Cloud Storage, BigQuery and Vertex AI.

Messaging Protocols

- **MQTT 3.1.1** and **5.0** over WebSocket and plain TCP.
- **AMQP 0.9.1** and **1.0** with RabbitMQ broker support.
- **STOMP** over WebSocket and TCP (ActiveMQ, RabbitMQ brokers).
- **WAMP 1.0** and **2.0** client and server protocols.
- **JSON-RPC 2.0** sub-protocol.

P2P and Real-Time Communication

- **WebRTC** server protocol.
- **STUN** and **TURN** client and server components.
- **ICE** client for NAT traversal.
- **UDP** client and server.

API Integrations

- **30+ built-in API connectors**: Binance, Coinbase, Kraken, Bitfinex, Discord, WhatsApp, Telegram, and more.
- **SignalR** and **SignalR Core**, **Socket.IO**, and **Pusher** clients.
- Cloud services: **Amazon SQS**, **Google Cloud Pub/Sub**, **Google Calendar**.

Cross-Platform

- **VCL** and **FireMonkey** (Windows, macOS, iOS, Android, Linux).
- **C++Builder** support.
- **Lazarus / FreePascal** support.
- Built-in **JavaScript libraries** for browser clients.

The following components are included in the sgcWebSockets library:

1. **sgcWebSockets**
   - **TsgcWebSocketClient:** WebSocket Client based on Indy Library.
   - **TsgcWebSocketServer:** WebSocket Server based on Indy Library
   - **TsgcWebSocketHTTPServer:** WebSocket + HTTP Server based on Indy Library.
   - **TsgcWebSocketServer_HTTPAPI:** Fast Performance WebSocket + HTTP Server based on HTTP.SYS Microsoft HTTP API.
   - **TsgcWebSocketClient_WinHTTP:** WebSocket Client based on WinHTTP Library.
   - **TsgcWebSocketFirewall:** WebSocket Server Firewall with IP filtering, brute force protection, rate limiting, and threat detection.
   - **TsgcWebSocketLoadBalancerServer:** Load Balancer Server for WebSocket and HTTP protocols.
   - **TsgcWebSocketProxyServer:** WebSocket Proxy Server allowing web browsers to connect to TCP servers.
2. **sgcWebSocket APIs**
   - **TsgcWSAPI_Binance:** Binance Spot Client, supports WebSocket + REST APIs.
   - **TsgcWSAPI_Binance_Futures:** Binance Futures Client, supports WebSocket + REST APIs.
   - **TsgcWSAPI_SocketIO:** Socket.IO Client.
   - **TsgcWSAPI_Coinbase:**Coinbase Pro Client, supports WebSocket + REST APIs.
   - **TsgcWSAPI_Bitmex:** Bitmex Client, supports WebSocket + REST APIs.
   - **TsgcWSAPI_SignalR:** SignalR WebSocket Client.
   - **TsgcWSAPI_SignalRCore:** SignalRCore WebSocket Client.
   - **TsgcWSAPI_Pusher:** Pusher WebSocket Client.
   - **TsgcWSAPI_Kraken:** Kraken Client API, supports WebSocket and REST Api.
   - **TsgcWSAPI_Kraken_Futures:** Kraken Futures Client API, supports WebSocket and REST Api.
   - **TsgcWSAPI_Bitstamp:** Bitstamp WebSocket Client.
   - **TsgcWSAPI_Cex:** Cex WebSocket Client.
   - **TsgcWSAPI_Huobi:** Huobi WebSocket Client.
   - **TsgcWSAPI_ThreeCommas:** ThreeCommas Client API.
   - **TsgcWSAPI_Bitfinex:** Bitfinex WebSocket API.
   - **TsgcWSAPI_Discord:** Discord WebSocket Client.
   - **TsgcWSAPI_BlockChain:** BlockChain WebSocket Client.
   - **TsgcWSAPI_Bybit:** Bybit WebSocket Client.
   - **TsgcWSAPI_OKX:** OKX WebSocket Client.
   - **TsgcWSAPI_Kucoin:** KuCoin Spot WebSocket Client.
   - **TsgcWSAPI_Kucoin_Futures:** KuCoin Futures WebSocket Client.

- **TsgcWSAPI_Deribit:** Deribit WebSocket Client.
- **TsgcWSAPI_CryptoCom:** Crypto.com WebSocket Client.
- **TsgcWSAPI_HTX:** HTX WebSocket Client.
- **TsgcWSAPI_MEXC:** MEXC WebSocket Client.
- **TsgcWSAPI_Bitget:** Bitget WebSocket Client.
- **TsgcWSAPI_GateIO:** Gate.io WebSocket Client.
- **TsgcWSAPI_XTB:** XTB WebSocket Client.
- **TsgcWSAPI_CexPlus:** CexPlus WebSocket Client.
- **TsgcWSAPI_OpenAI:** OpenAI Realtime WebSocket API Client.

3. **sgcWebSocket Libs**
   - **TsgcTDLib_Telegram:** Telegram API Client.
   - **TsgcWhatsApp_Client:** WhatsApp Business Cloud Client.
   - **TsgcHTTP_Cryptohopper:** Cryptohopper Client API.
   - **TsgcLib_RCON:** RCON Client.
4. **sgcWebSocket Protocols**
   - **TsgcWSPClient_MQTT:** MQTT (3.1.1 and 5.0) Client. Supports WebSocket and Plain TCP Connections.
   - **TsgcWSPClient_AMQP1:** AMQP 1.0.0 Client. Supports RabbitMQ Brokers.
   - **TsgcWSPClient_AMQP:** AMQP 0.9.1 Client. Supports RabbitMQ Brokers.
   - **TsgcWSPClient_STOMP:** STOMP Client, supports WebSocket and Plain TCP Connections.
     - **TsgcWSPClient_STOMP_ActiveMQ:** STOMP Client for ActiveMQ Broker.
     - **TsgcWSPClient_STOMP_RabbitMQ:** STOMP Client for RabbitMQ Broker.
   - **TsgcWSPClient_WAMP:** WAMP 1.0 Client Protocol.
   - **TsgcWSPServer_WAMP:** WAMP 1.0 Server Protocol.
   - **TsgcWSPClient_WAMP2:** WAMP 2.0 Client Protocol.
   - **TsgcWSPServer_WebRTC:** WebRTC Server Protocol.
   - **TsgcWSPClient_sgc:** WebSocket Client SGC Protocol based on JSON RPC.
   - **TsgcWSPServer_sgc:** WebSocket Server SGC Protocol based on JSON RPC.
   - **TsgcWSPClient_Files:** WebSocket File Transfer Client Protocol.
   - **TsgcWSPServer_Files:** WebSocket File Transfer Server Protocol.
   - **TsgcWSPClient_Dataset:** WebSocket Client Dataset Synchronization Protocol.
   - **TsgcWSPServer_Dataset:** WebSocket Server Dataset Synchronization Protocol.
   - **TsgcWSPClient_Presence:** WebSocket Client Presence Protocol.
   - **TsgcWSPServer_Presence:** WebSocket Server Presence Protocol.
   - **TsgcWSPClient_E2EE:** End-To-End Encryption Client Protocol.
   - **TsgcWSPServer_E2EE:** End-To-End Encryption Server Protocol.
5. **sgcWebSockets HTTP**
   - **TsgcHTTP1Client:**HTTP 1.0 Client based on Indy TIdHTTP.
   - **TsgcHTTP2Client:** HTTP 2.0 Client.

- **TsgcGRPCClient:** gRPC Client over HTTP/2, with typed interfaces for Google Cloud services (Pub/Sub, Speech-to-Text, Translation, Vision, Natural Language, Cloud Storage, BigQuery, Vertex AI).
- **TsgcSSEClient:** Server-Sent Events (EventSource) Client with automatic reconnection and Last-Event-ID resume.
- **TsgcHTTP_JWT_Client:** JWT (JSON WEB TOKEN) Client.
- **TsgcHTTP_JWT_Server:** JWT (JSON WEB TOKEN) Server.
- **TsgcHTTP_OAuth2_Client:** OAuth 2.0 Client.
- **TsgcHTTP_OAuth2_Server:** OAuth 2.0 Server.
- **TsgcHTTPAWS_SQS_Client:** Amazon AWS SQS Client.
- **TsgcHTTPGoogleCloud_PubSub_Client:** Google Cloud Pub/Sub Client.
- **TsgcHTTPGoogleCloud_Calendar_Client:** Google Calendar Client.
- **TsgcHTTP_OAuth2_Client_Google:** OAuth 2.0 Client for Google.
- **TsgcHTTP_OAuth2_Client_Microsoft:** OAuth 2.0 Client for Microsoft.
- **TsgcHTTP_OAuth2_Server_Provider:** OAuth 2.0 Server Provider.
- **TsgcWSAPIServer_WebAuthn:** WebAuthn Server for passkey-based authentication.
- **TsgcTOTPAuthenticator:** TOTP and HOTP one-time passwords for two-factor authentication (Enterprise).
- **TsgcLDAPClient:** LDAP v3 and Active Directory Client (Enterprise).
- **TsgcSAMLServiceProvider:** SAML 2.0 single sign-on Service Provider for Entra ID, Okta, AD FS and other identity providers (Enterprise).
- **TsgcHTTP_OIDC_Client:** OpenID Connect Client with discovery, PKCE and ID token validation (Enterprise).
- **TsgcMailOAuth2:** OAuth 2.0 for SMTP, IMAP and POP3 with Microsoft 365 and Gmail (Enterprise).
- **TsgcHTTPGoogleCloud_FCM_Client:** Google Cloud FCM (Firebase Cloud Messaging) Client.

6 **sgcWebSockets IoT**

- **TsgcIoTAmazon_MQTT_Client:** Amazon MQTT IoT Core Client.
- **TsgcIoTAzure_MQTT_Client:** Azure IoT MQTT Client.

7 **sgcWebSockets P2P**

- **TsgcUDPCLient:** UDP Client.
- **TsgcUDPServer:** UDP Server.
- **TsgcSTUNClient:** STUN Client.
- **TsgcSTUNServer:** STUN Server.
- **TsgcTURNClient:** STUN / TURN Client.
- **TsgcTURNServer:** STUN / TURN Server.
- **TsgcICEClient:** ICE Client.
- **TsgcRTCPeerConnection:** WebRTC Peer Connection Client.

8 **sgcWebSockets DataSnap**

- **TsgcWSHTTPWebBrokerBridgeServer:** DataSnap Server Replacement with HTTP + WebSockets Support.

- **TsgcWSHTTP2WebBrokerBridgeServer:** DataSnap Server Replacement with HTTP + HTTP/2 + WebSockets Support.
- **TsgcWSServer_HTTPAPI_WebBrokerBridge:** DataSnap Server Replacement based on HTTP.SYS Microsoft Server.

9 **sgcWebSockets AI**

- **TsgcAIOpenAIChatBot:** Build a ChatBot with Voice Commands.
- **TsgcAIOpenAITranslator:** Real-Time translation.
- **TsgcAudioRecorderMCI:** Record Audio using MCI.
- **TsgcAudioPlayerMCI:** Play Audio using MCI.
- **TsgcTextToSpeechSystem:** Text-To-Speech using operating system default.
- **TsgcTextToSpeechGoogle:** Text-To-Speech using Google Cloud.
- **TsgcTextToSpeechAmazon:** Text-To-Speech using Amazon AWS.
- **TsgcAIOpenAIEmbeddings:** allows you to use your custom data to build AI applications.
- **TsgcAIDatabaseVectorFile:** stores the vectors in a plain text file.
- **TsgcAIDatabaseVectorPinecone:** supports the Pinecone vector database.
- **TsgcAIOpenAIAssistant:** OpenAI Assistants API Client.
- **TsgcAIDocumentExtractor:** Extracts structured data from PDF documents and images.
- **TsgcAIOCRTesseract:** Local OCR based on Tesseract.
- **TsgcAIOCRWindows:** Local OCR built into Windows 10 and later.
- **TsgcAIVoiceAgent:** Realtime voice agent with OpenAI Realtime and Gemini Live.
- **TsgcWSAPIServer_MCP:** MCP (Model Context Protocol) Server.
- **TsgcWSAPIClient_MCP:** MCP (Model Context Protocol) Client.

Versions Support

Delphi supported IDE

- Delphi 7 (* only supported if upgraded to Indy 10, Intraweb is not supported)
- Delphi 2007
- Delphi 2009
- Delphi 2010
- Delphi XE
- Delphi XE2
- Delphi XE3
- Delphi XE4
- Delphi XE5
- Delphi XE6
- Delphi XE7
- Delphi XE8
- Delphi 10 Seattle
- Delphi 10.1 Berlin
- Delphi 10.2 Tokyo
- Delphi 10.3 Rio
- Delphi 10.4 Sydney
- Delphi 11 Alexandria
- Delphi 12 Athens
- Delphi 13 Florence

CBuilder supported IDE

- CBuilder 2007
- CBuilder 2010
- CBuilder XE
- CBuilder XE2
- CBuilder XE3
- CBuilder XE4
- CBuilder XE5
- CBuilder XE6
- CBuilder XE7
- CBuilder XE8
- CBuilder 10 Seattle
- CBuilder 10.1 Berlin
- CBuilder 10.2 Tokyo
- CBuilder 10.3 Rio
- CBuilder 10.4 Sydney
- CBuilder 11 Alexandria
- CBuilder 12 Athens
- CBuilder 13 Florence

FreePascal supported IDE

- Lazarus

Trial Version

Compiled *.dcu files provided with the free version use the default Indy and Intraweb versions. If you have upgraded any of these packages, they probably will not work and you will need to buy the full source code version.

Indy Package

Some components use Indy as their TCP/IP library (such as TsgcWebSocketClient or TsgcWebSocketServer), which means that Indy is needed in order to install the sgcWebSockets package. By default, sgcWebSockets uses the Indy library built into RAD Studio, but we provide a custom Indy version that has more features: support for OpenSSL API 1.1, OpenSSL 3.0, ALPN protocol...

Installation

Delphi / CBuilder / Lazarus

1. Unzip the included files into a directory {$DIR}

2. From Delphi\CBuilder:

Add the directory where the files are unzipped {$DIR} to the Delphi\CBuilder library path under Tools, Environment options, Directories

All Delphi\CBuilder Versions

Add the directory {$DIR}\source to the library path

For specific Delphi version

Delphi 7 : Add the directory {$DIR}\libD7 to the library path
Delphi 2007 : Add the directory {$DIR}\libD2007 to the library path
Delphi 2009 : Add the directory {$DIR}\libD2009 to the library path
Delphi 2010 : Add the directory {$DIR}\libD2010 to the library path
Delphi XE : Add the directory {$DIR}\libDXE to the library path
Delphi XE2 : Add the directory {$DIR}\libDXE2\$(Platform) to the library path
Delphi XE3 : Add the directory {$DIR}\libDXE3\$(Platform) to the library path
Delphi XE4 : Add the directory {$DIR}\libDXE4\$(Platform) to the library path
Delphi XE5 : Add the directory {$DIR}\libDXE5\$(Platform) to the library path
Delphi XE6 : Add the directory {$DIR}\libDXE6\$(Platform) to the library path
Delphi XE7 : Add the directory {$DIR}\libDXE7\$(Platform) to the library path
Delphi XE8 : Add the directory {$DIR}\libDXE8\$(Platform) to the library path
Delphi 10 : Add the directory {$DIR}\libD10\$(Platform) to the library path
Delphi 10.1 : Add the directory {$DIR}\libD10_1\$(Platform) to the library path
Delphi 10.2 : Add the directory {$DIR}\libD10_2\$(Platform) to the library path
Delphi 10.3 : Add the directory {$DIR}\libD10_3\$(Platform) to the library path
Delphi 10.4 : Add the directory {$DIR}\libD10_4\$(Platform) to the library path
Delphi 11 : Add the directory {$DIR}\libD11\$(Platform) to the library path
Delphi 12 : Add the directory {$DIR}\libD12\$(Platform) to the library path
Delphi 13 : Add the directory {$DIR}\libD13\$(Platform) to the library path

For specific CBuilder version

C++ Builder 2010 : Add the directory {$DIR}\libD2010 to the library path
C++ Builder XE : Add the directory {$DIR}\libDXE to the library path
C++ Builder XE2 : Add the directory {$DIR}\libDXE2\$(Platform) to the library path
C++ Builder XE3 : Add the directory {$DIR}\libDXE3\$(Platform) to the library path
C++ Builder XE4 : Add the directory {$DIR}\libDXE4\$(Platform) to the library path
C++ Builder XE5 : Add the directory {$DIR}\libDXE5\$(Platform) to the library path
C++ Builder XE6 : Add the directory {$DIR}\libDXE6\$(Platform) to the library path
C++ Builder XE7 : Add the directory {$DIR}\libDXE7\$(Platform) to the library path
C++ Builder XE8 : Add the directory {$DIR}\libDXE8\$(Platform) to the library path
C++ Builder 10 : Add the directory {$DIR}\libD10\$(Platform) to the library path
C++ Builder 10.1 : Add the directory {$DIR}\libD10_1\$(Platform) to the library path
C++ Builder 10.2 : Add the directory {$DIR}\libD10_2\$(Platform) to the library path
C++ Builder 10.3 : Add the directory {$DIR}\libD10_3\$(Platform) to the library path
C++ Builder 10.4 : Add the directory {$DIR}\libD10_4\$(Platform) to the library path
C++ Builder 11 : Add the directory {$DIR}\libD11\$(Platform) to the library path
C++ Builder 12 : Add the directory {$DIR}\libD12\$(Platform) to the library path

C++ Builder 13 : Add the directory {$DIR}\libD13\$(Platform) to the library path

For all CBuilder versions, Add dcp\$(Platform) to the library path (contains .bpi files)

3. From Delphi

Choose

File, Open and browse for the correct Packages\sgcWebSockets.groupproj (First compile sgcWebSocketsX.dpk and then install dclsgcWebSocketsX.dpk)

packages files for Delphi

sgcWebSocketsD7.groupproj : Delphi 7
sgcWebSocketsD2007.groupproj : Delphi 2007
sgcWebSocketsD2009.groupproj : Delphi 2009
sgcWebSocketsD2010.groupproj : Delphi 2010
sgcWebSocketsDXE.groupproj : Delphi XE
sgcWebSocketsDXE2.groupproj : Delphi XE2
sgcWebSocketsDXE3.groupproj : Delphi XE3
sgcWebSocketsDXE4.groupproj : Delphi XE4
sgcWebSocketsDXE5.groupproj : Delphi XE5
sgcWebSocketsDXE6.groupproj : Delphi XE6
sgcWebSocketsDXE7.groupproj : Delphi XE7
sgcWebSocketsDXE8.groupproj : Delphi XE8
sgcWebSocketsD10.groupproj : Delphi 10
sgcWebSocketsD10_1.groupproj : Delphi 10.1
sgcWebSocketsD10_2.groupproj : Delphi 10.2
sgcWebSocketsD10_3.groupproj : Delphi 10.3
sgcWebSocketsD10_4.groupproj : Delphi 10.4
sgcWebSocketsD11.groupproj : Delphi 11
sgcWebSocketsD12.groupproj : Delphi 12
sgcWebSocketsD13.groupproj : Delphi 13

4. From CBuilder

Choose

File, Open and browse for the correct Packages\sgcWebSockets.groupproj (First compile sgcWebSocketsX.dpk and then install dclsgcWebSocketsX.dpk)

packages files for CBuilder

sgcWebSocketsC2010.groupproj : C++ Builder 2010
sgcWebSocketsCXE.groupproj : C++ Builder XE
sgcWebSocketsCXE2.groupproj : C++ Builder XE2
sgcWebSocketsCXE3.groupproj : C++ Builder XE3
sgcWebSocketsCXE4.groupproj : C++ Builder XE4
sgcWebSocketsCXE5.groupproj : C++ Builder XE5
sgcWebSocketsCXE6.groupproj : C++ Builder XE6
sgcWebSocketsCXE7.groupproj : C++ Builder XE7
sgcWebSocketsCXE8.groupproj : C++ Builder XE8
sgcWebSocketsC10.groupproj : C++ Builder 10
sgcWebSocketsC10_1.groupproj : C++ Builder 10.1
sgcWebSocketsC10_2.groupproj : C++ Builder 10.2
sgcWebSocketsC10_3.groupproj : C++ Builder 10.3
sgcWebSocketsC10_4.groupproj : C++ Builder 10.4
sgcWebSocketsC11.groupproj : C++ Builder 11
sgcWebSocketsC12.groupproj : C++ Builder 12
sgcWebSocketsC13.groupproj : C++ Builder 13

5. From Lazarus

Choose : File, Open and browse Packages\sgcWebSocketsLazarus.lpk (First compile and then install)

Compiled files are located in the Lazarus directory. Inside it, there is an Indy directory with the latest Indy source version.

Tested with Lazarus 2.0.6 and Indy 10.5.9.4930

6. Demos

All demos are available in the Demos subdirectory. Just open the project and run it. Intraweb demos may require modifying some units due to different Intraweb versions.

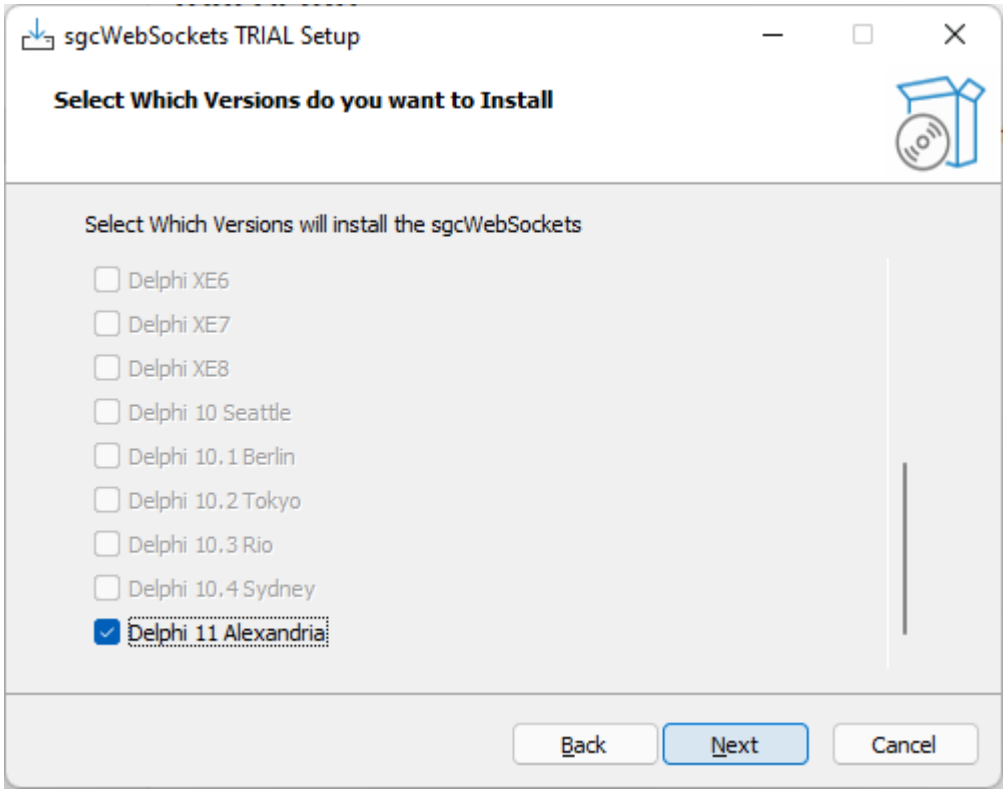
Install Setup

*Requires Windows Vista as minimum (Windows 2000, XP and Server 2003 are not supported).

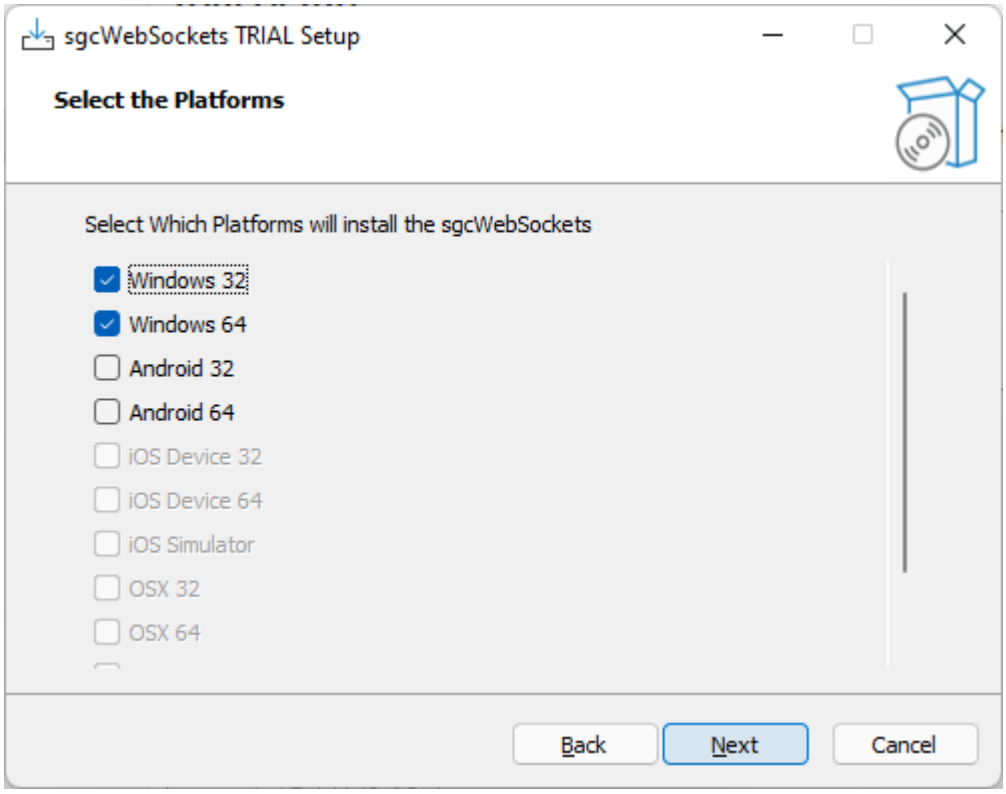
If you use the Windows Setup to install sgcWebSockets library, the installation is guided and very simple. If there is any error while installing, please refer to Install Errors page and you can try to install the package manually.

Trial Setup

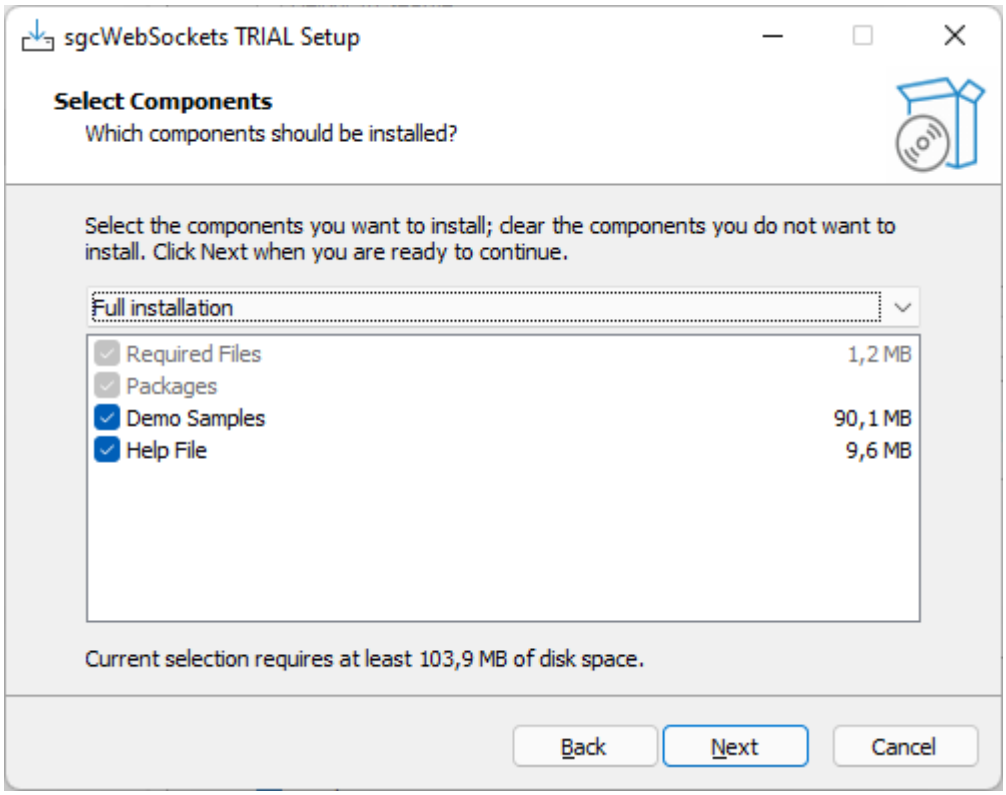
- Execute the Trial Installer.
- The Trial setup requires Admin privileges.
- The installer will show a list of Delphi / CBuilder / Rad Studio versions and by default the downloaded version will be enabled. If this version is NOT detected by the installer, the installer will extract the files but won't try to compile. Please refer to install the package manually.



- The next page shows the Platforms that can be installed, only those platforms detected by the installer are enabled.



- The next page shows the license agreement which must be accepted to install the trial.
- After accepting the license agreement, it shows the components that will be installed. By default, the package, compiled DCUs, demos, and help files will all be installed. You can customize whether the help files and demos are installed or not.

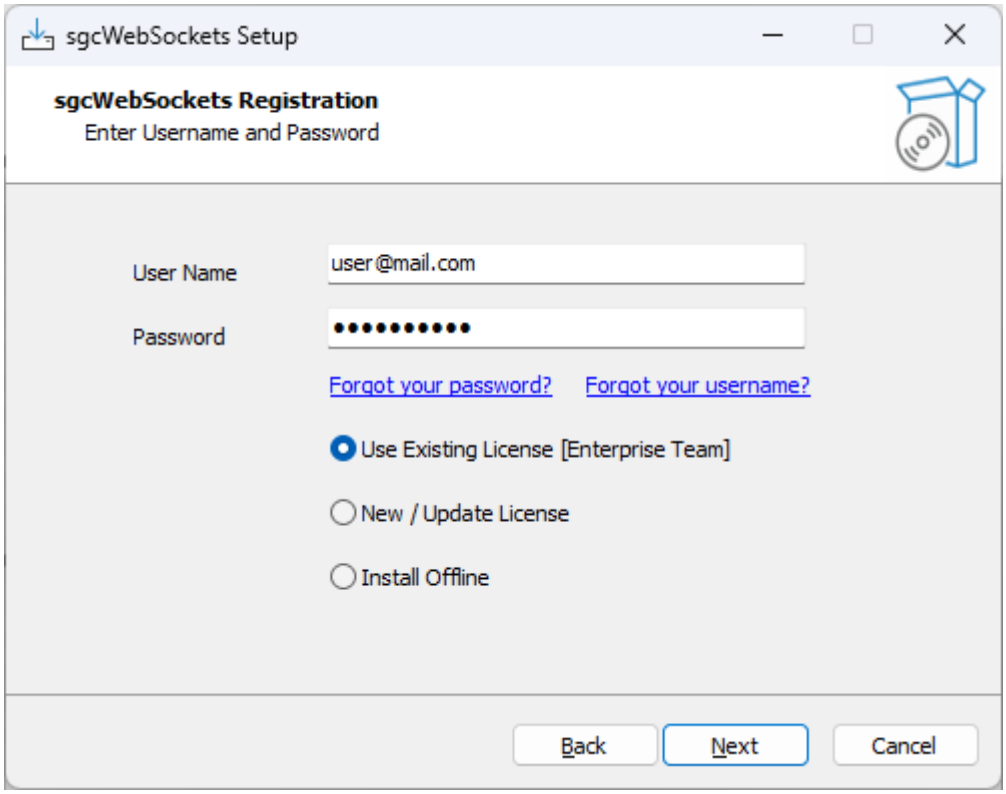


- Finally, it will extract the files, compile and install the package and register the required paths in the IDE.

Customers Setup

Users who have purchased a license can install the sgcWebSockets Library using the setup. Find below the step-by-step instructions for installing the package.

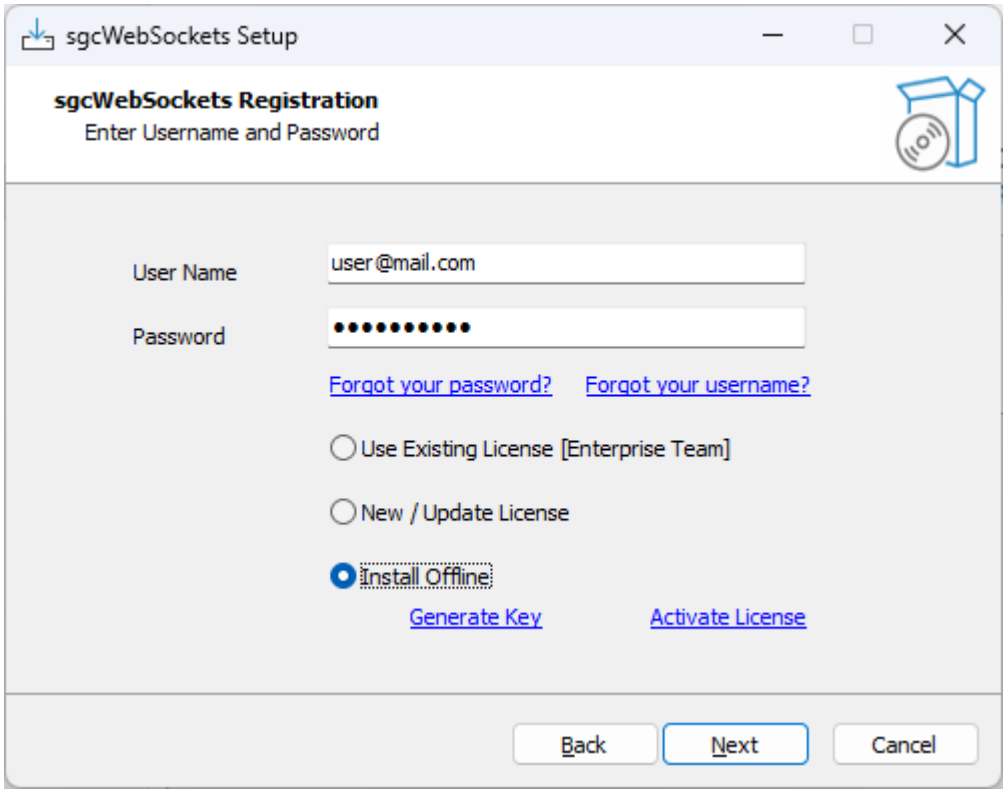
- Execute the Installer.
- The installer runs with the lowest privileges (if it runs as admin, it cannot be installed on network drives). If the destination requires admin privileges, run the setup as administrator.
- First you must set the username/password of your private eSeGeCe account. This only needs to be entered once; the next time you use the setup, the installer will read the previously saved value.



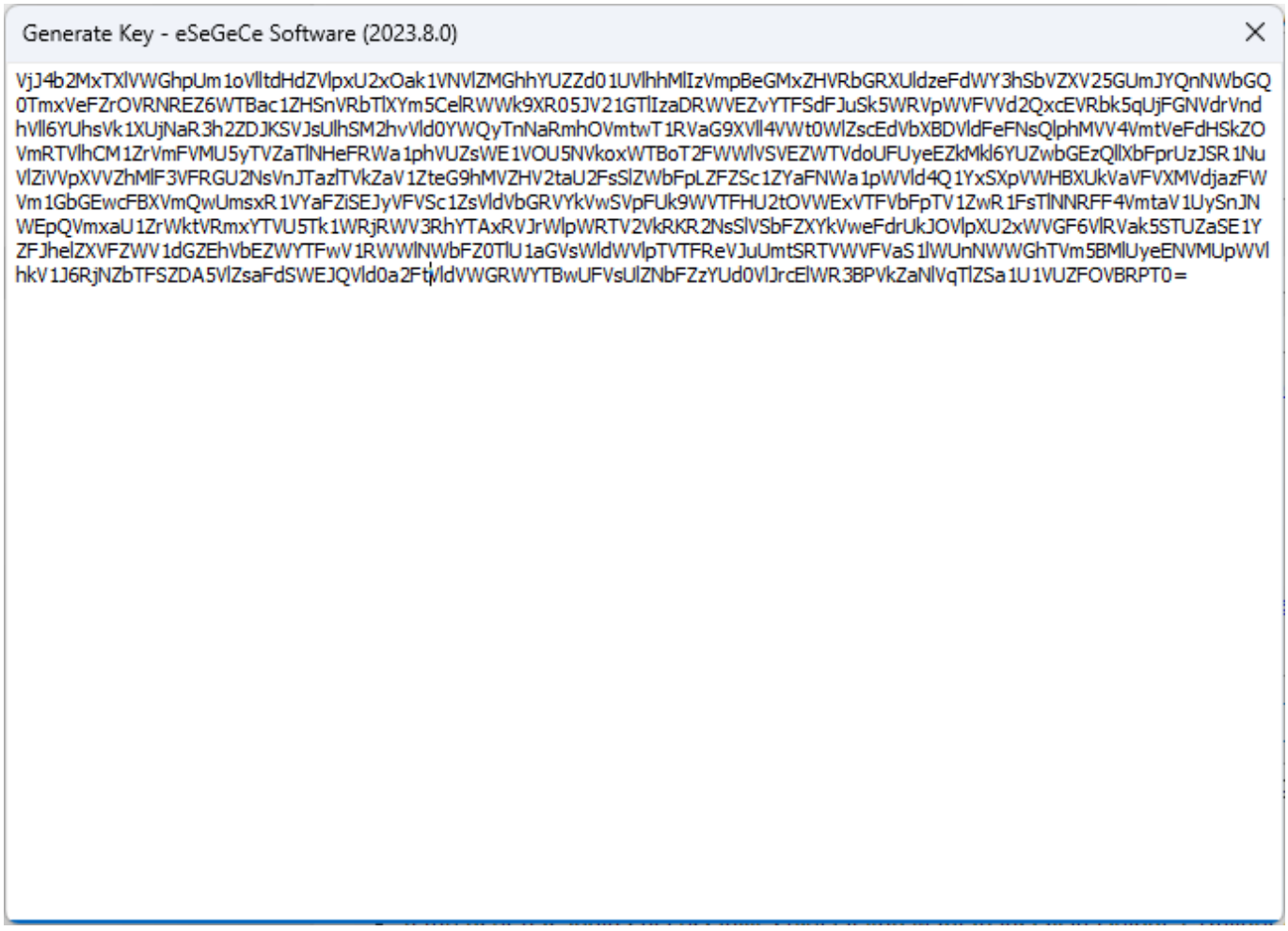
- There are 3 options:
  - **Use Existing License:** if this version has been already installed, the option will be selected by default. It will use the latest configuration for this version.
  - **New / Update License:** if this version has not been installed previously, this option will be selected by default. It will connect to the Server License to get the license information. If you have upgraded your license recently, you can select this option to update the license.
  - **Install Offline:** if the machine does not have internet access, select this option to activate your license.

Generate Key

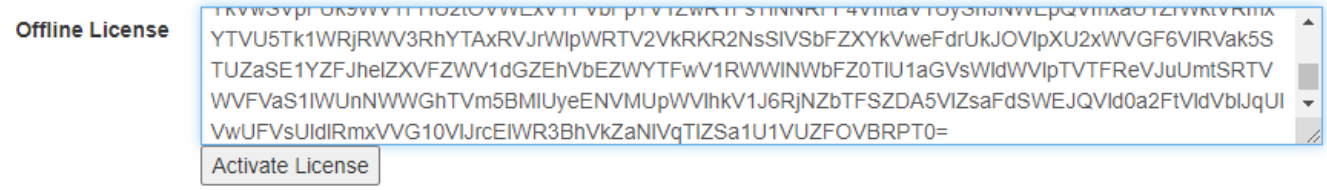
This option generates a key that will be used to activate the license.



Copy the key and access to your private online account: www.esegece.com/my-account/subscriptions.

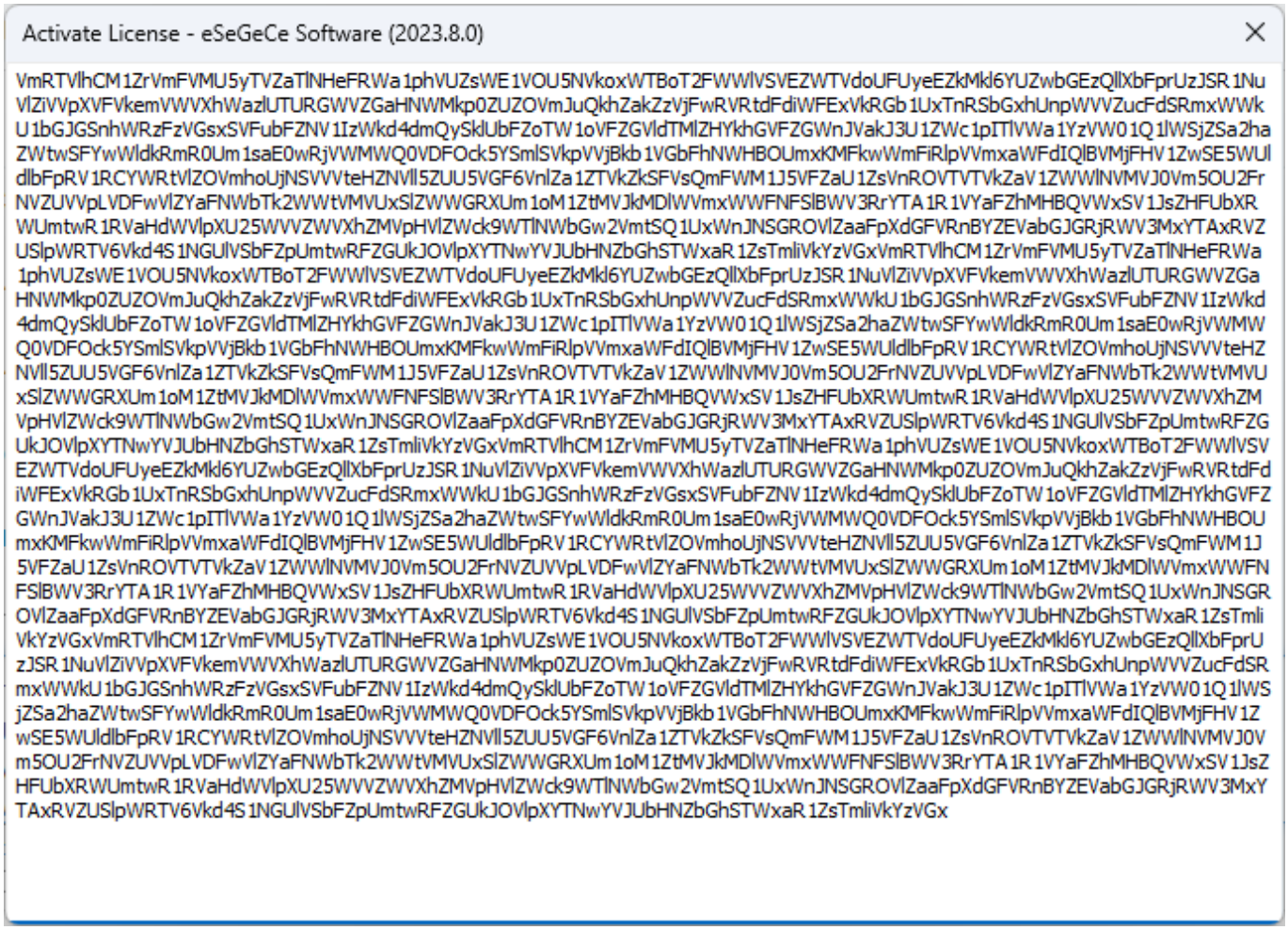


Select the subscription to activate and paste the key.

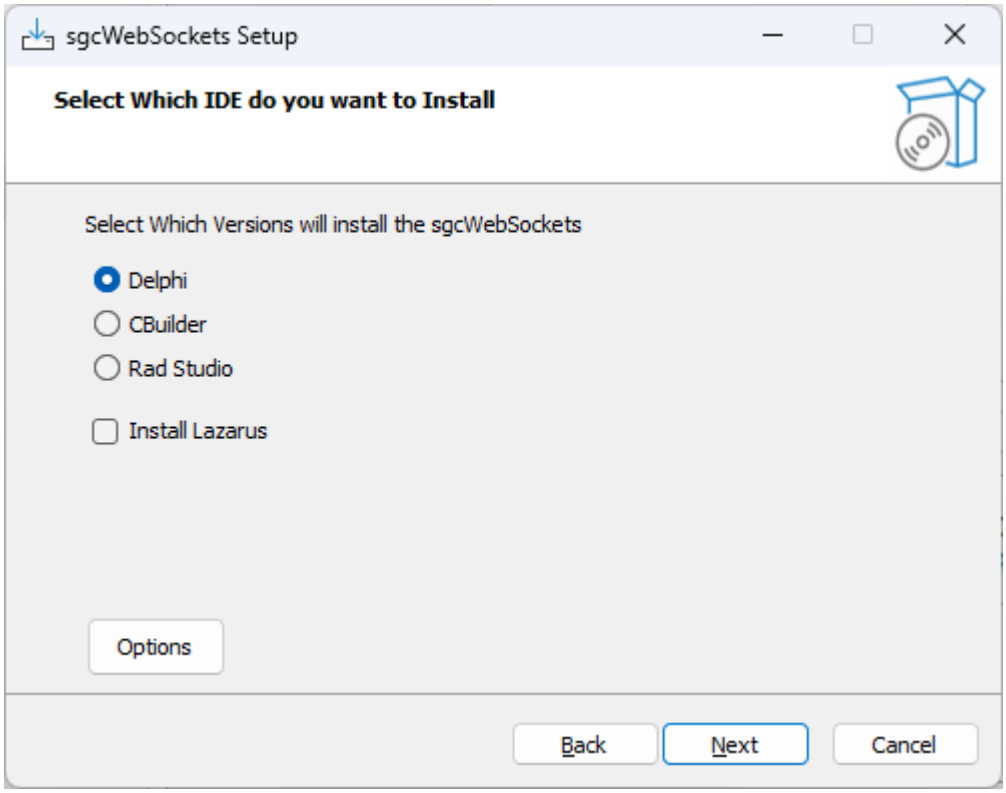


Activate License

If the request is correct it will return a license that must be copied in the setup.

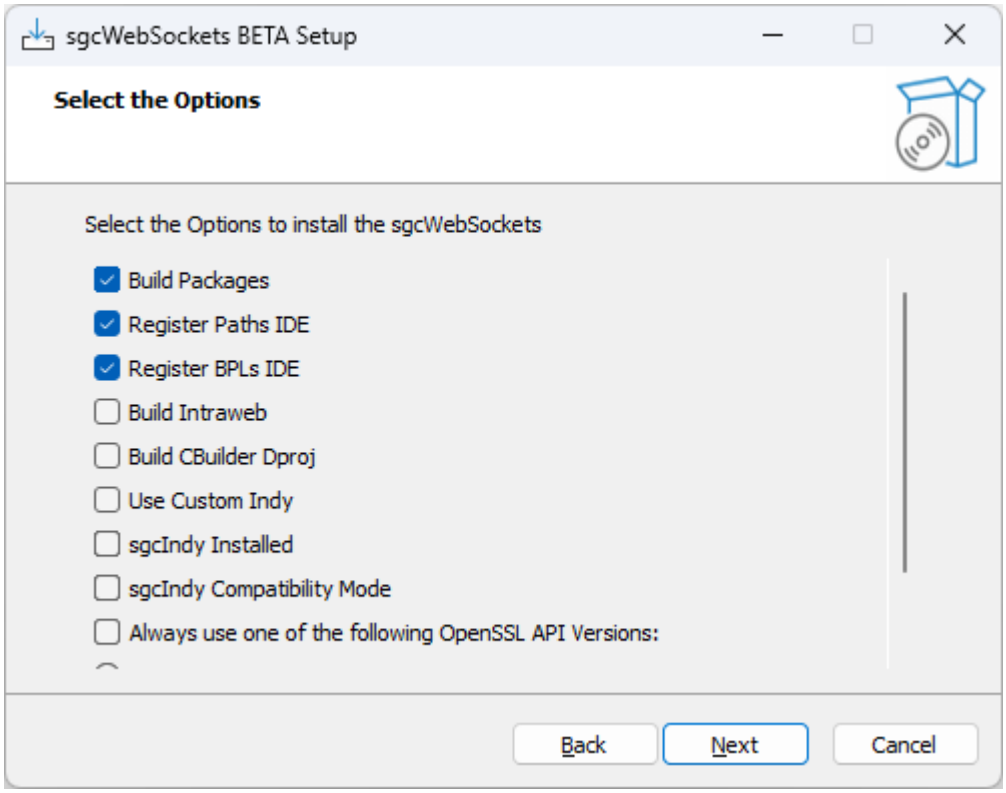


- If the license has been activated successfully, select if you want to install in Delphi, CBuilder or Rad Studio IDE. There is a check to extract the required lazarus files (Lazarus requires to install the package manually).

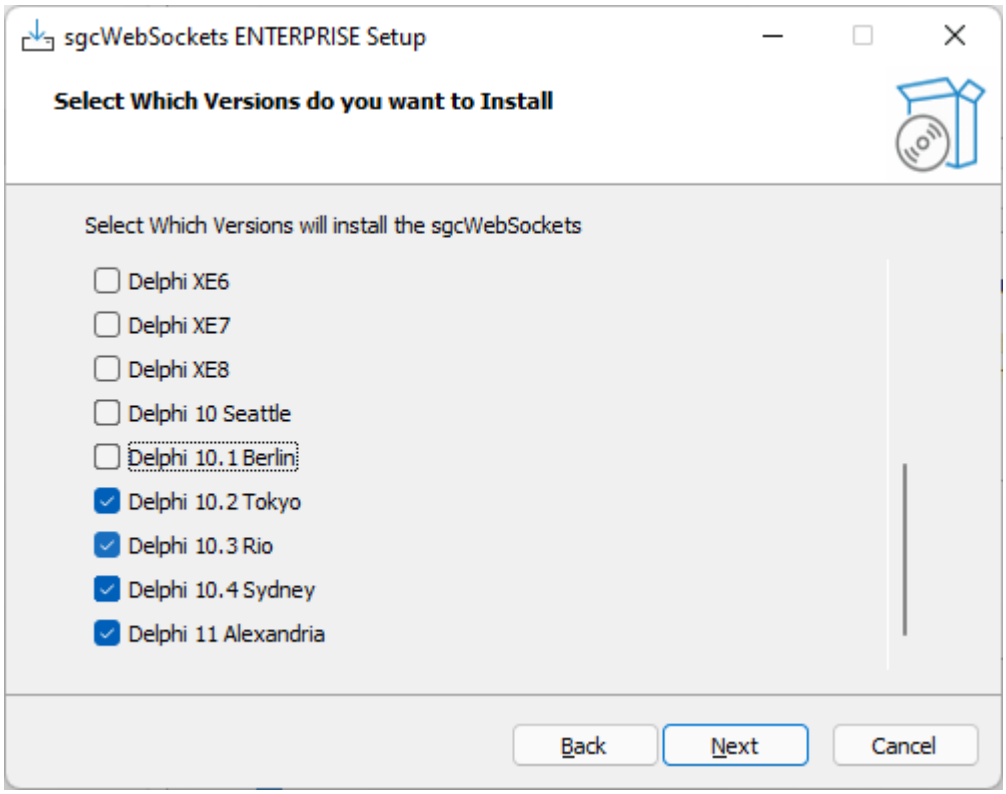


- There are some options that can be customized every time you use the installer, press the button **Options** to access these properties.
  - **Build Packages:** if selected, the installer will try to build the packages.
  - **Register Paths IDE:** if selected, the installer will register the required library paths in the IDE.
  - **Register BPLs IDE:** if selected and the installer has built the packages successfully, the installer will register the design-time package in the IDE.

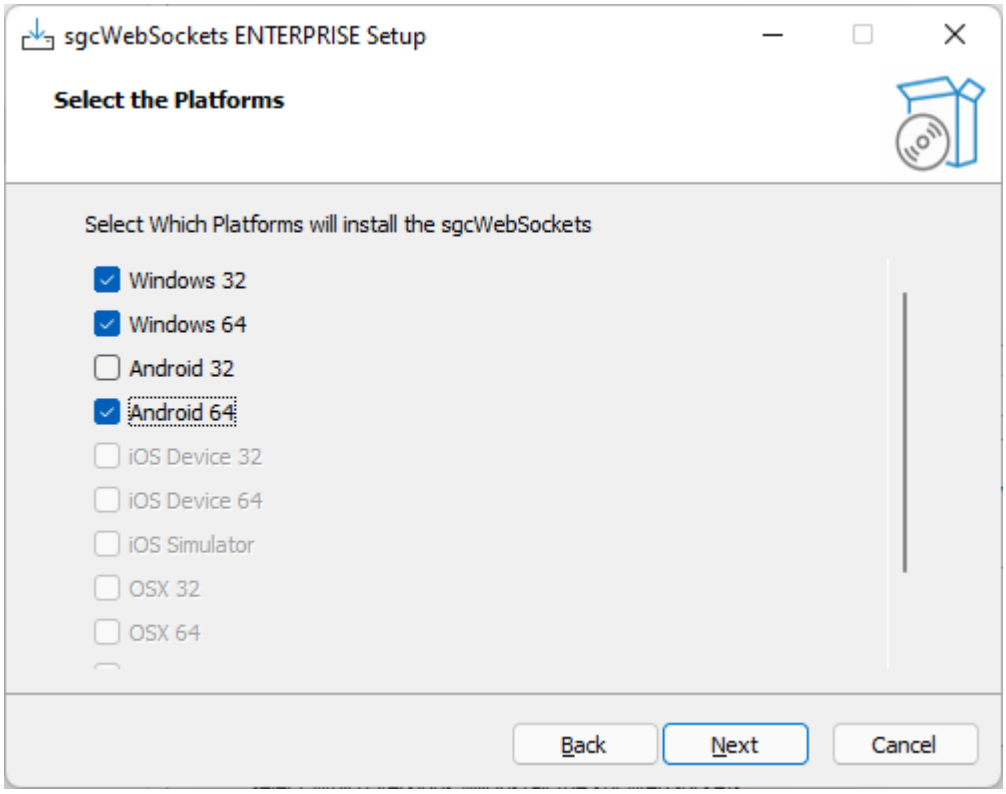
- The following options are only available for licenses with source code:
  - **Build Intraweb:** if selected, the installer will install the required Intraweb files (disabled by default). Currently installs the Intraweb XVI version.
  - **Build CBuilder Dproj:** if selected, the installer will build the CBuilder package using the sgcWebSockets Delphi package and generating all required CBuilder files.
  - **Use Custom Indy:** (only Enterprise), if selected, the sgcWebSockets will use the Custom Indy Version (with support for openSSL 1.1 and 3.0, TLS 1.3, ALPN...)
  - **sgcIndy Already Installed:** if the sgcIndy package has been installed and you want to use this package to compile sgcWebSockets package, check this option.
  - **sgcIndy Compatibility Mode:** if the sgcIndy package has been installed in Compatibility Mode (because other packages are using Indy, like DevExpress), check this option.
  - **Always use of the following OpenSSL API Versions:** check this option if you want to force the use of OpenSSL 1.1.1 or OpenSSL 3.0.0 APIs
  - **Debug Mode:** saves debug messages in a log file. Do not use this mode in a production environment.
  - **Build Rad Studio IDE Win64:** builds the design time package for Rad Studio IDE 12+ 64bits.
  - **Include Resources:** enabled by default. Embeds the JavaScript client bundle (`sgcResources.RES`) into the runtime packages. The embedded resource is only needed if your server delivers the sgcWebSockets browser client over HTTP to remote browsers. Uncheck this option to undefine `SGC_RESOURCES` in `sgcVer.inc` before the packages are compiled, reducing the size of every executable that links against the source. Safe to disable for Delphi / C++Builder client-only applications and for servers that don't serve the bundled JS client. See Configure Install for the underlying compiler directive.



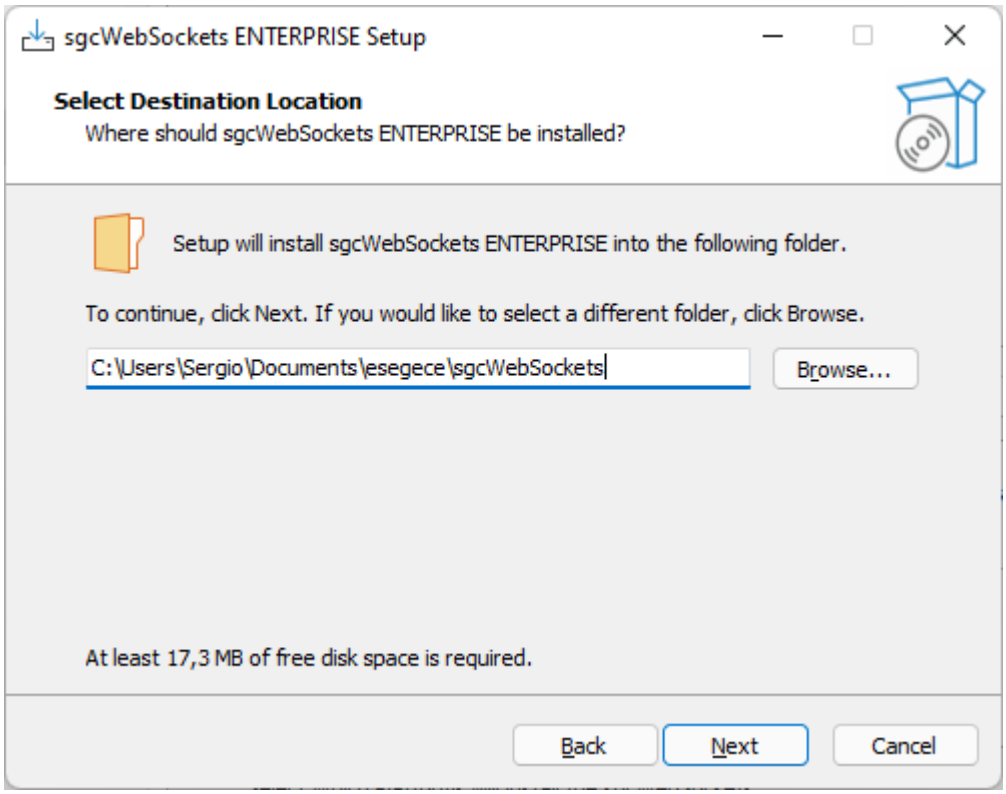
- Now you can select which IDE versions you want to install. Only those IDE versions that the installer detects as installed will be available.



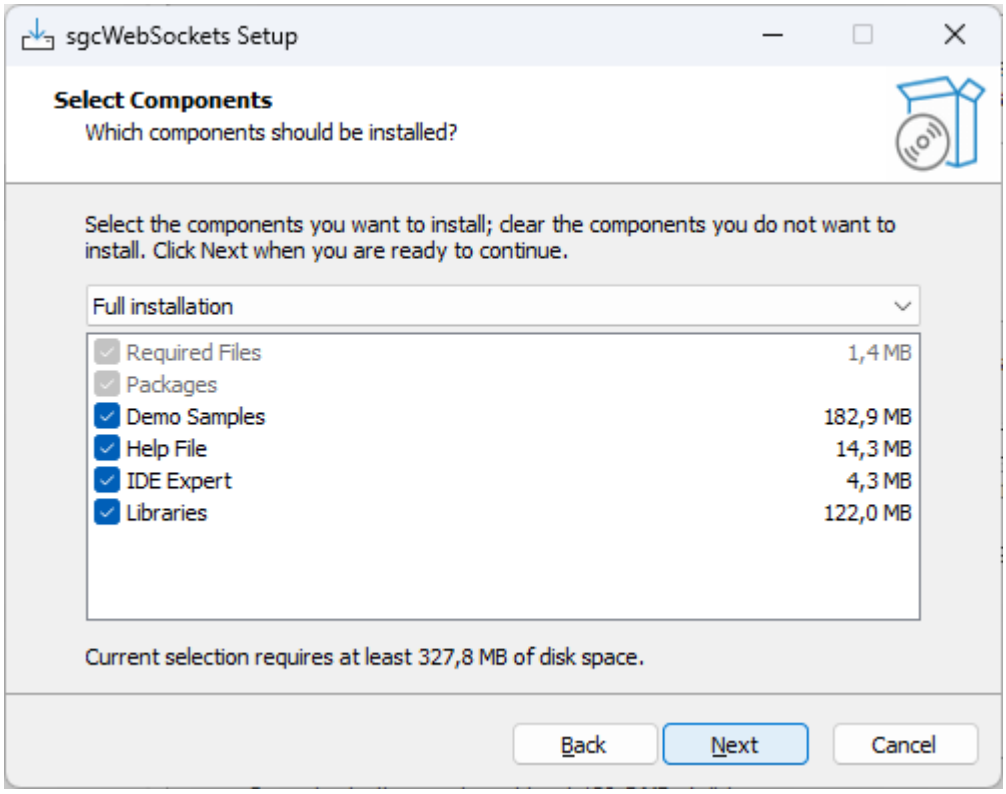
- The next step is to select the platforms.



- Select the folder where the package will be installed. If you reinstall the package, the installer will select by default the same folder selected in the previous install.



- Select which components to install. The registered customers have an **IDE expert** that allows you to connect to the eSeGeCe account from the IDE, know if there are available updates, direct access to helpdesk... and more.



- Finally, it will extract the files, compile and install the package and register the required paths in the IDE.

Install Errors

- MsBuild raises an error if the Length of the **Library Path is too high**, to fix this issue, try to delete unused paths from the library path. MsBuild has a limitation of 32K characters.
- error **F2110: 'tchar.h' file not found**. This error may appear when compiling for CBuilder, to fix this, just open CBuilder, close it and try to install again.

Install Command Line Parameters

The following commands are supported by the installer.

/SILENT
The wizard and the background window are not displayed but the installation progress is

/VERYSILENT
When a setup is very silent this installation progress window is not displayed.

/EXTRACT
The package is not installed only extracted. The path where it's installed can be customized using /EXTRACT=path-to-folder
Use this parameter and /SILENT if you only want to extract the files without user interaction.

/IDE
This parameter allows you to set which IDE you want to install. Set one of the following:

- delphi
- cbuilder
- radstudio

Additionally you can add Lazarus.

Example: install delphi and lazarus.

```
/ide=delphi-lazarus.
```

/VERSIONS
Using this parameter you can set which RAD Studio versions you want to install. Multiple options are allowed:

- D7
- D2007
- D2009
- D2010
- DXE
- DXE2
- DXE3
- DXE4
- DXE5
- DXE6
- DXE7
- DXE8
- D10
- D10_1
- D10_2
- D10_3
- D10_4
- D11
- D12
- D13

Use the value "All" to install all possible versions.

Example: install Delphi 10 and Delphi 13.

```
/versions=D10-D13
```

/PLATFORMS
Using this parameter you can set which RAD Studio platforms you want to install. Multiple options are allowed:

- Win32
- Win64
- Win64x
- Android
- Android64
- iOSDevice32
- iOSDevice64
- iOSSimulator
- iOSSimARM64
- OSX32
- OSX64
- OSXARM64
- Linux64

Use the value "All" to install all possible platforms.

Example: install Win32 and Win64.

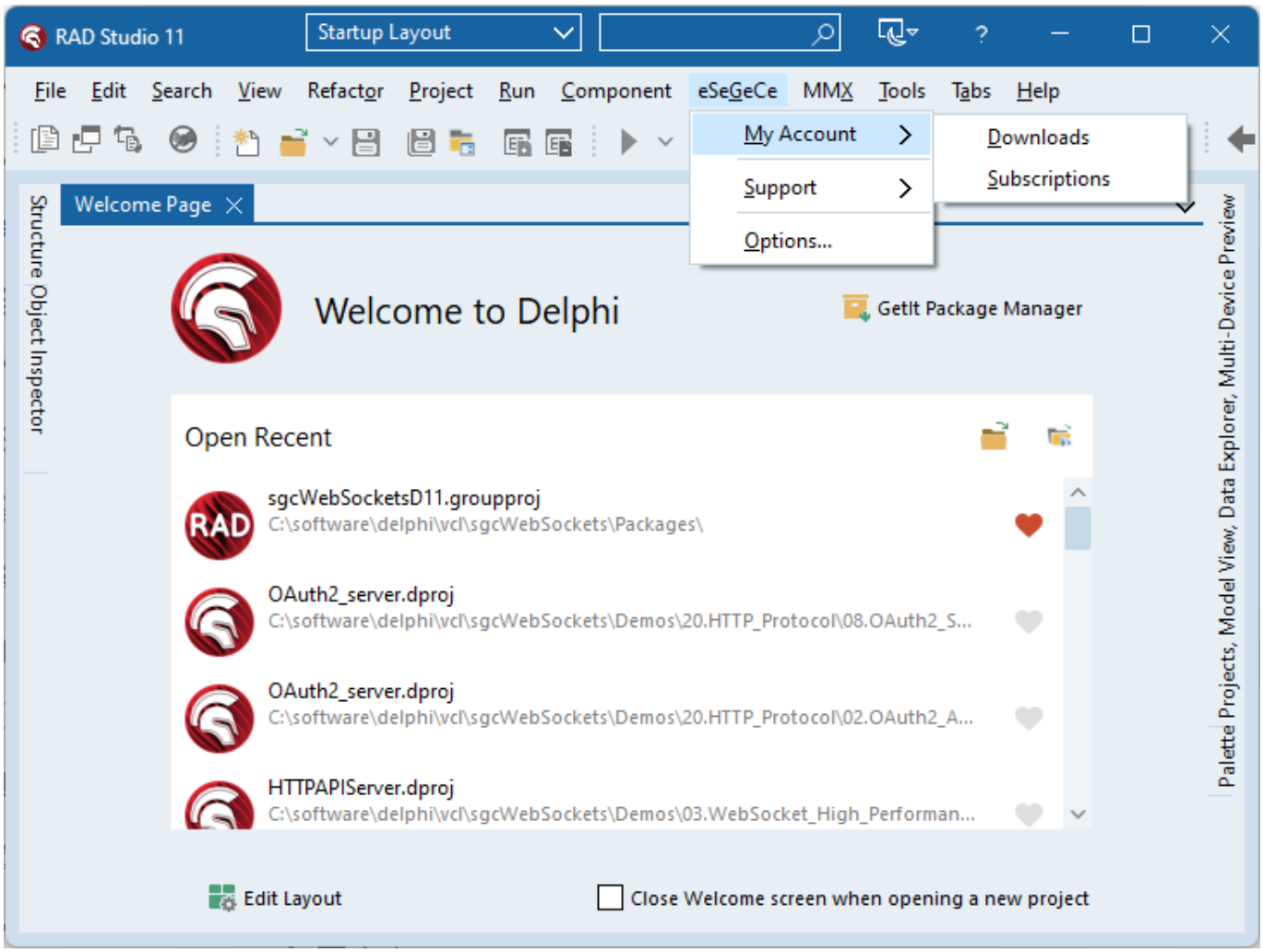
```
/platforms=Win32-Win64
```

/USERNAME
Sets the username of the subscription

/PASSWORD
Sets the password of the subscription

IDE Expert

If the IDE Expert is installed, you will find the following menu options:

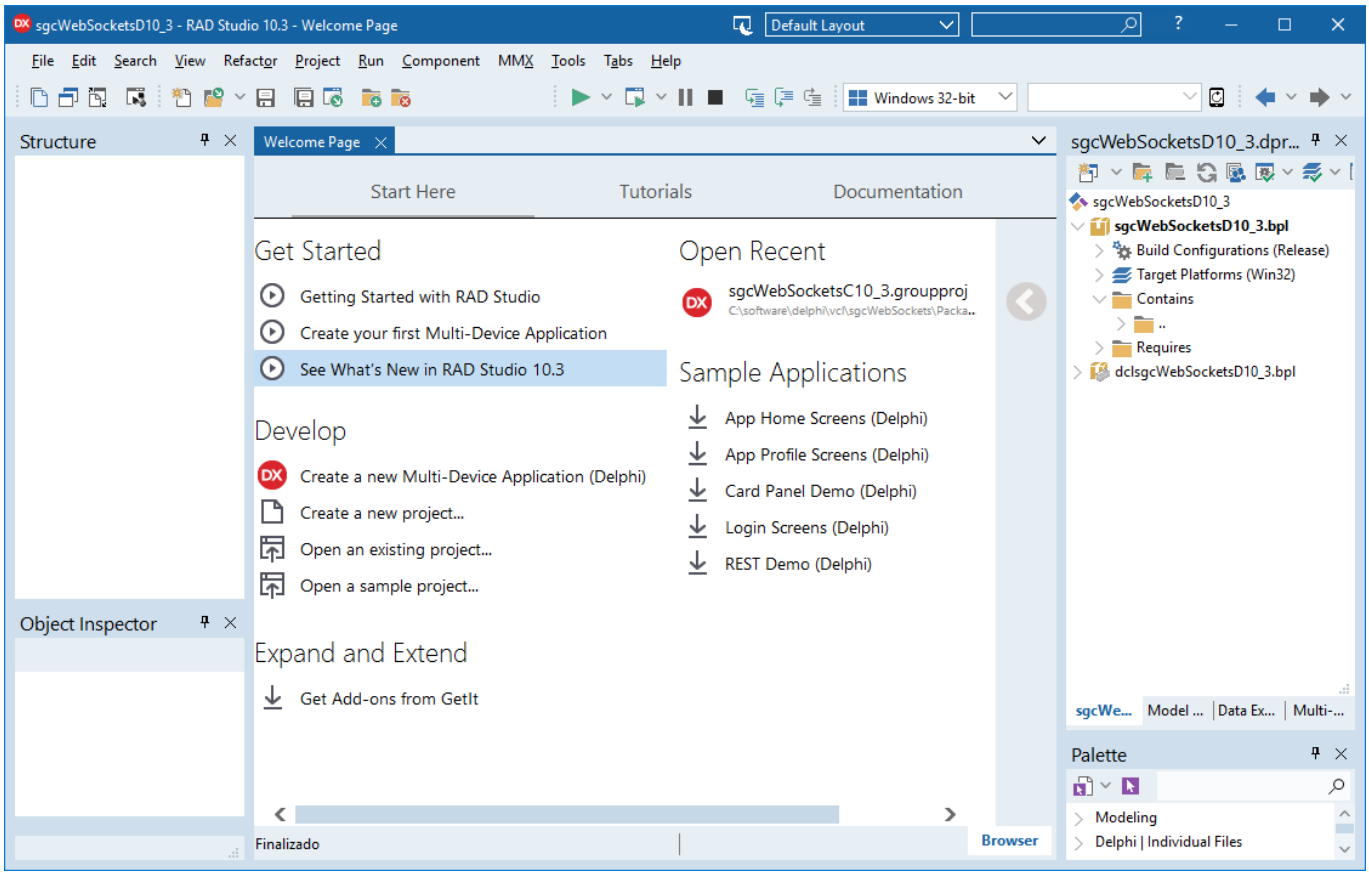


- **MyAccount:** direct access to the Downloads menu (where you can download the latest version of beta) and to the Subscriptions, to manage your license or renew an expired license.
- **Support:** direct access to HelpDesk or Forum with automatic login. Documentation and Contact Us form is available too.
- **Options:** in this menu you can configure the username/password of your account. Select the default browser and check if there are any updates available.

Install Package Manually

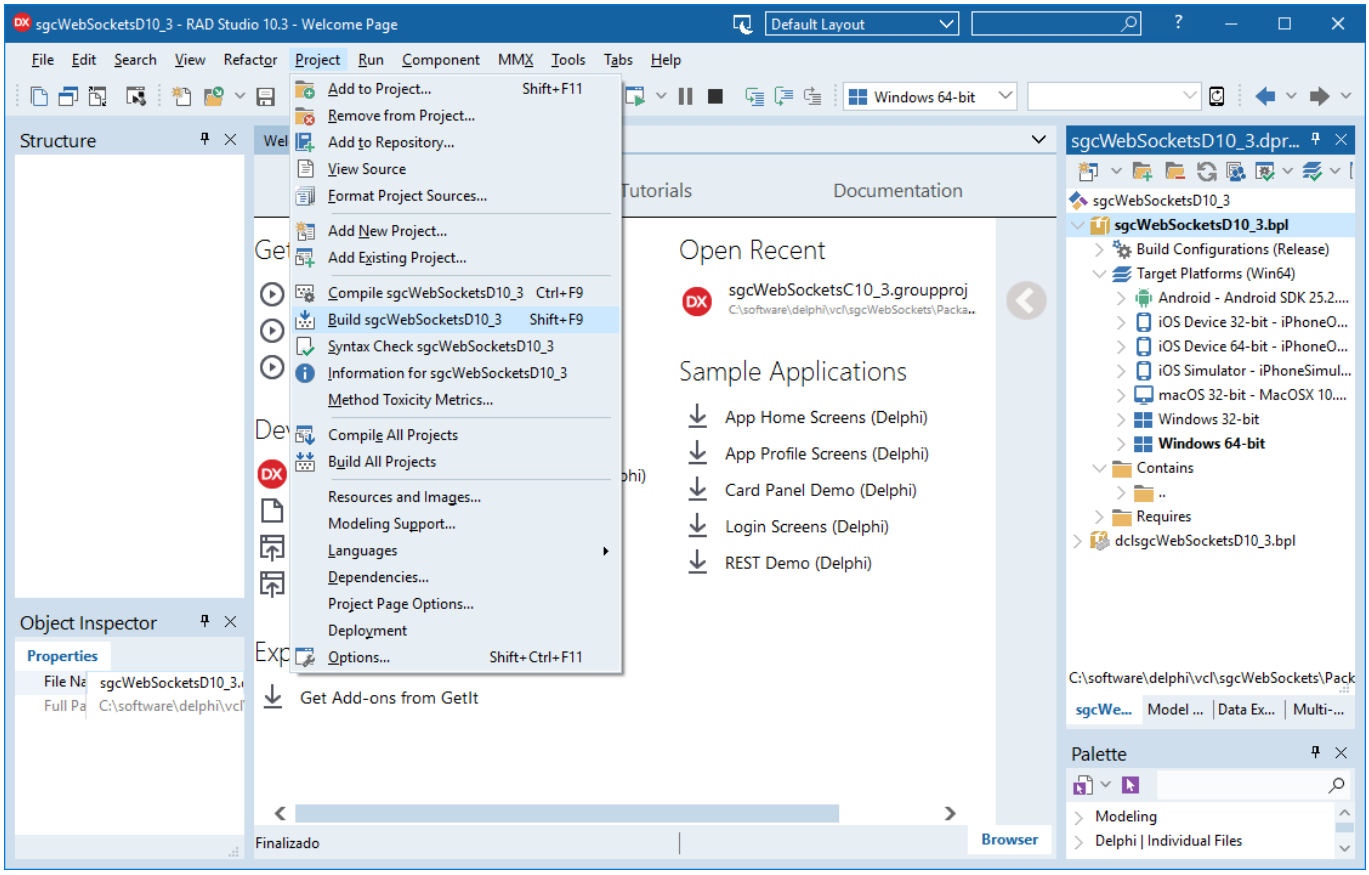
Follow next steps to install sgcWebSockets package, screenshots use Delphi 10.3 version.

1. Open sgcWebSocketsD10_3 group project.

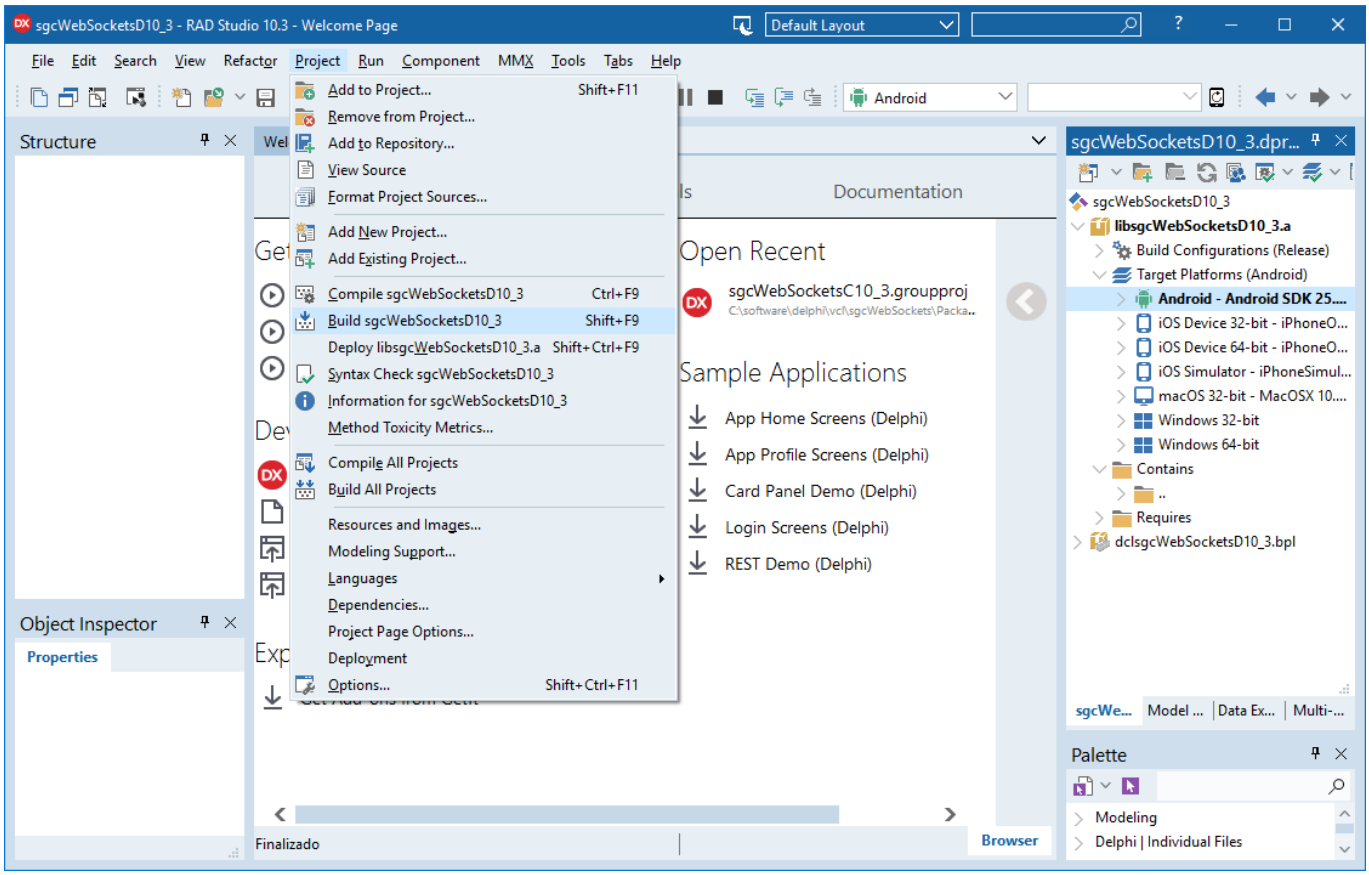


2. Now we must **compile first runtime packages** (name starts with sgcWebSockets). There is one package for every target platform and this depends on the Delphi version, so **select target platform one by one and build** every package.

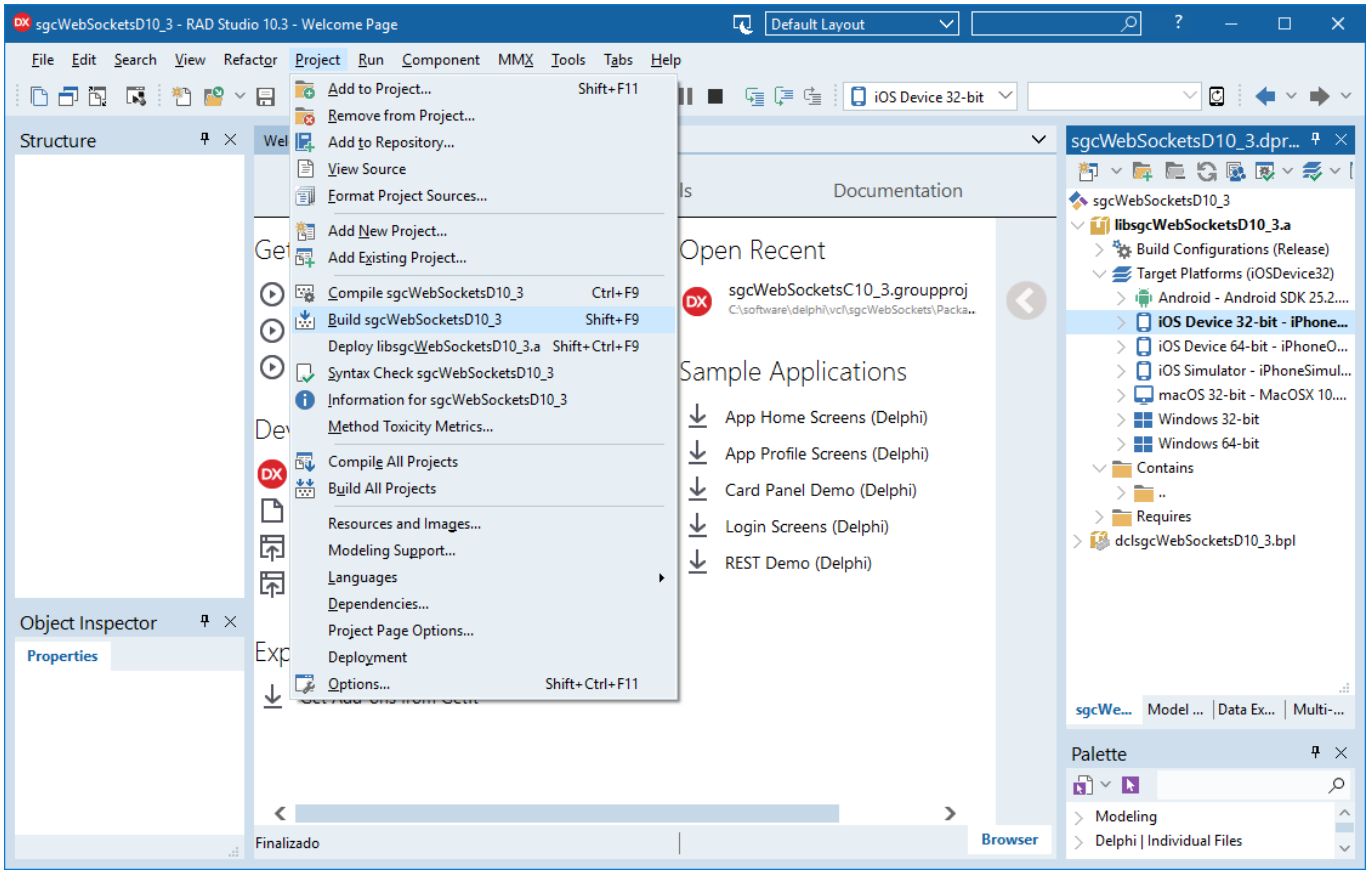
3. Select win64 as Target platform and build package.



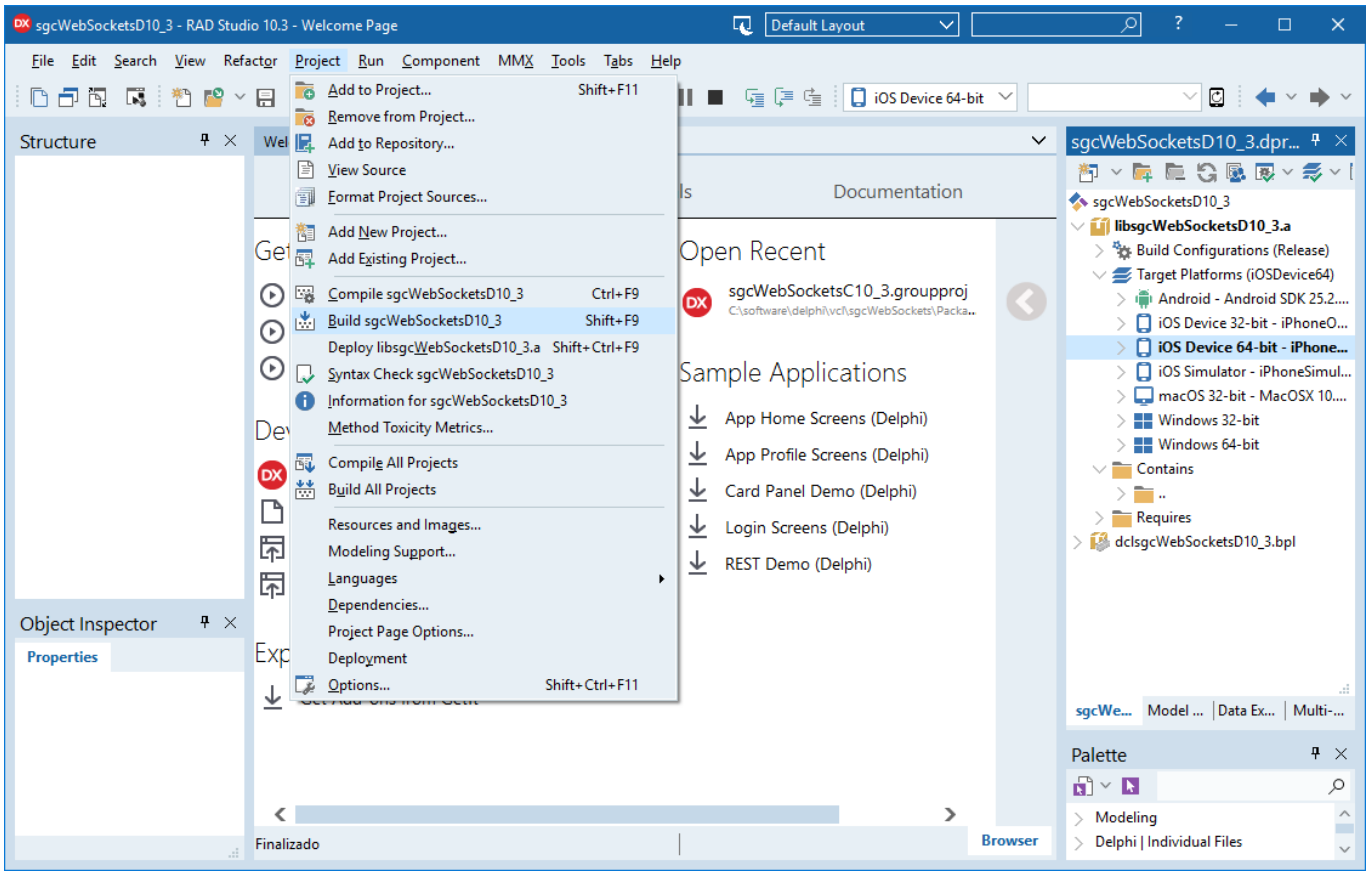
4. Select Android as Target Platform and build package.



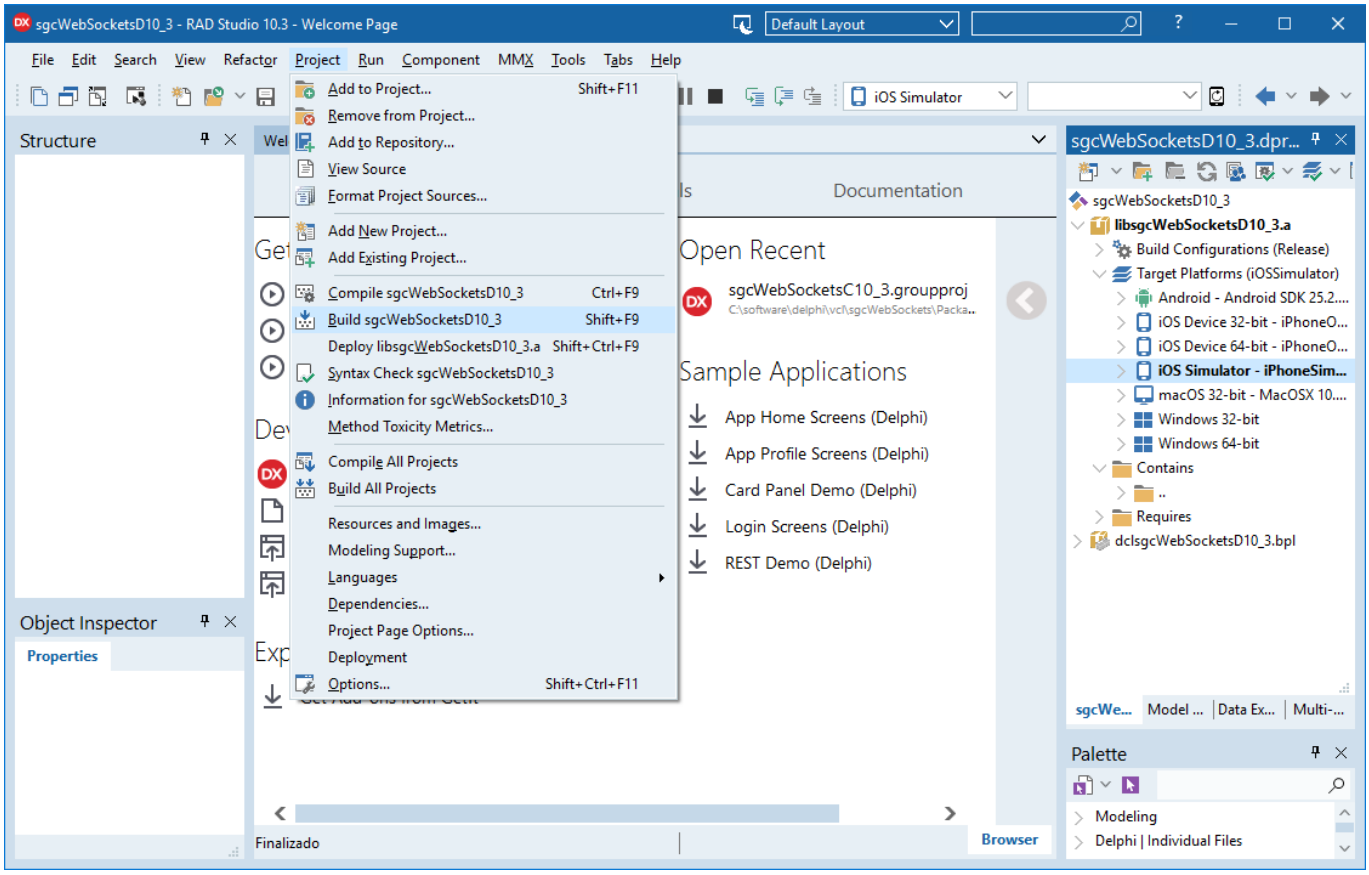
5. Select **iOS Device 32** as Target Platform and build package.



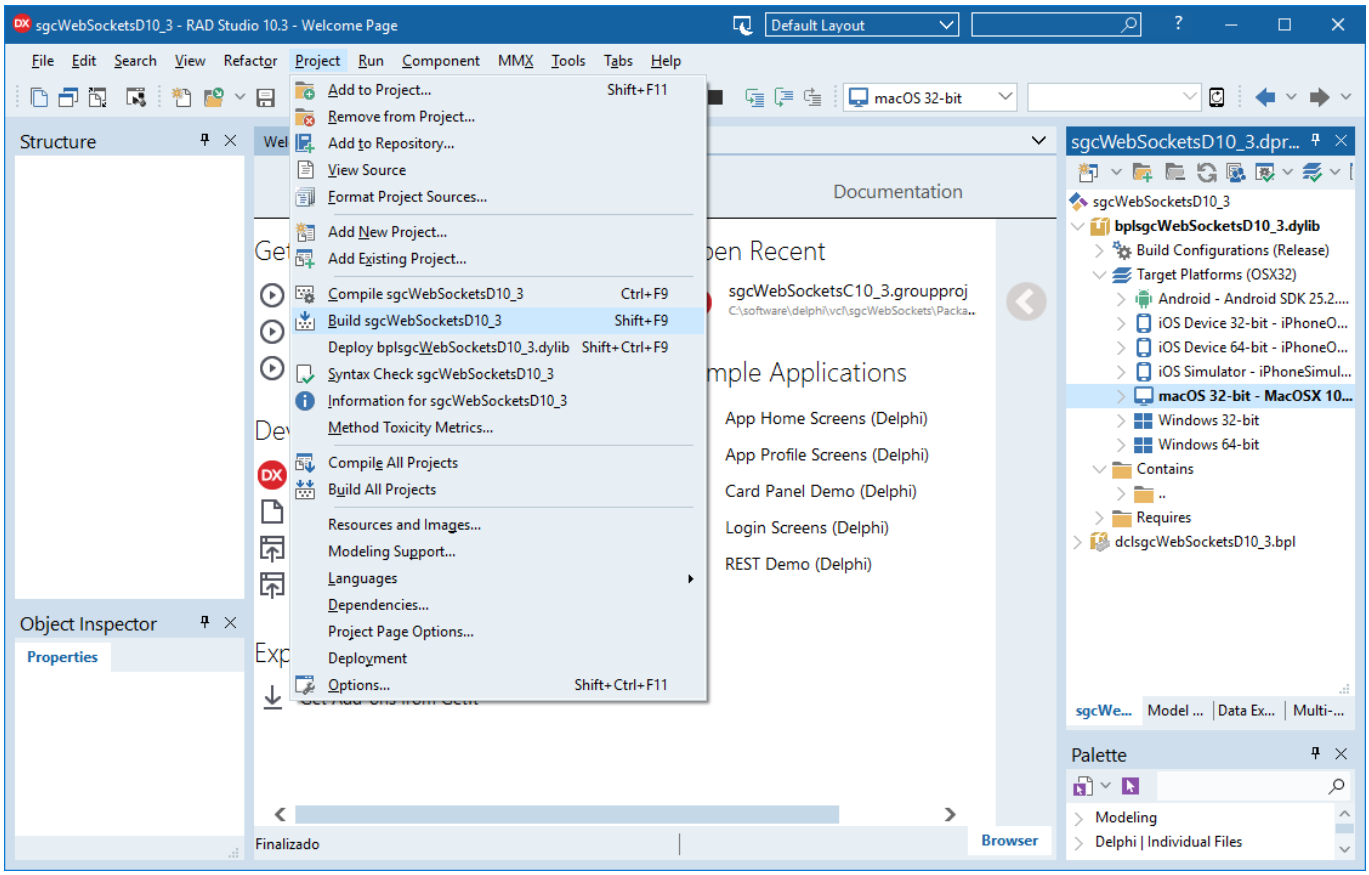
6. Select **iOS Device 64** as Target Platform and build package.



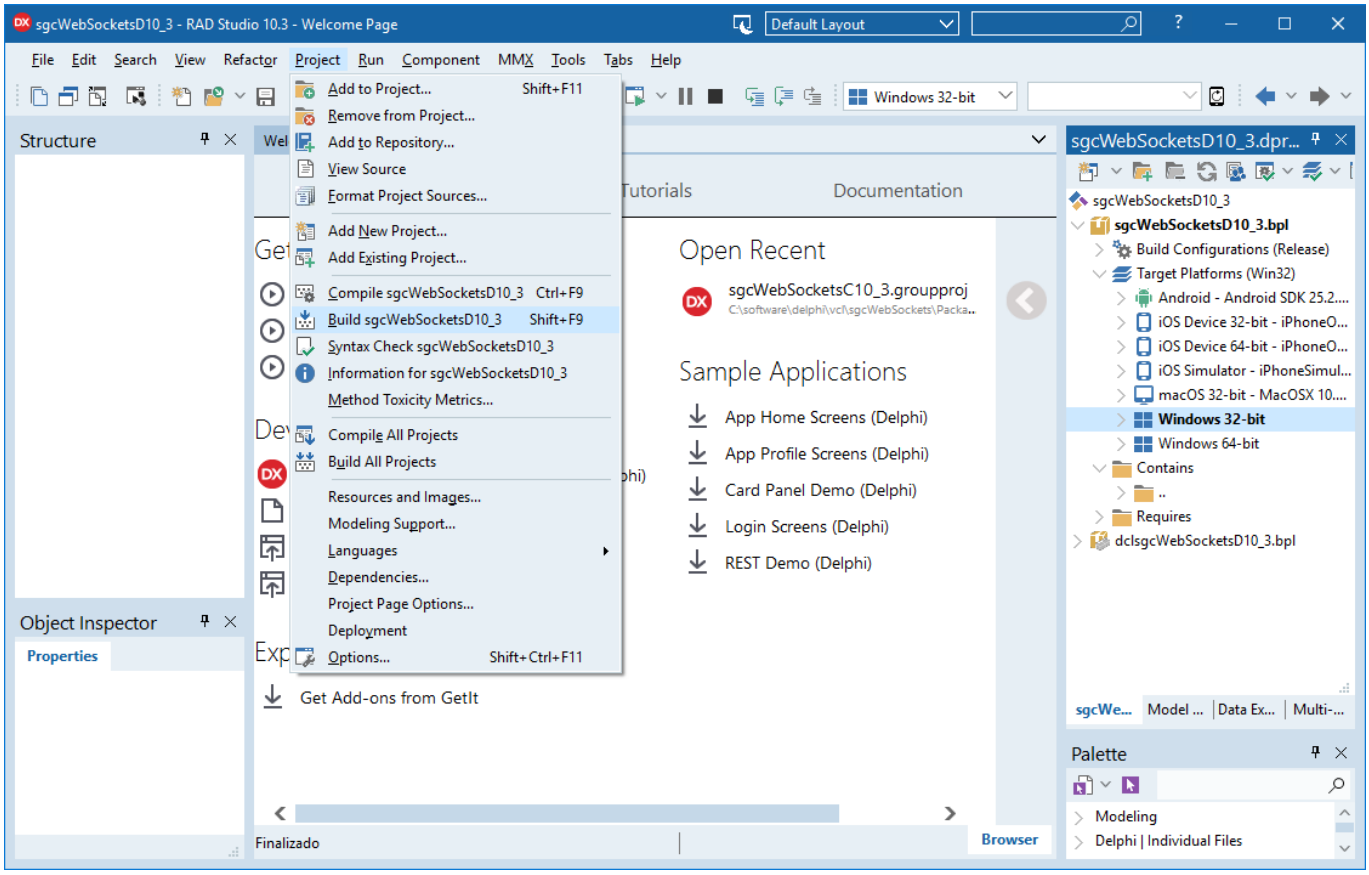
7. Select **iOS Device** Simulator as Target Platform and build package.



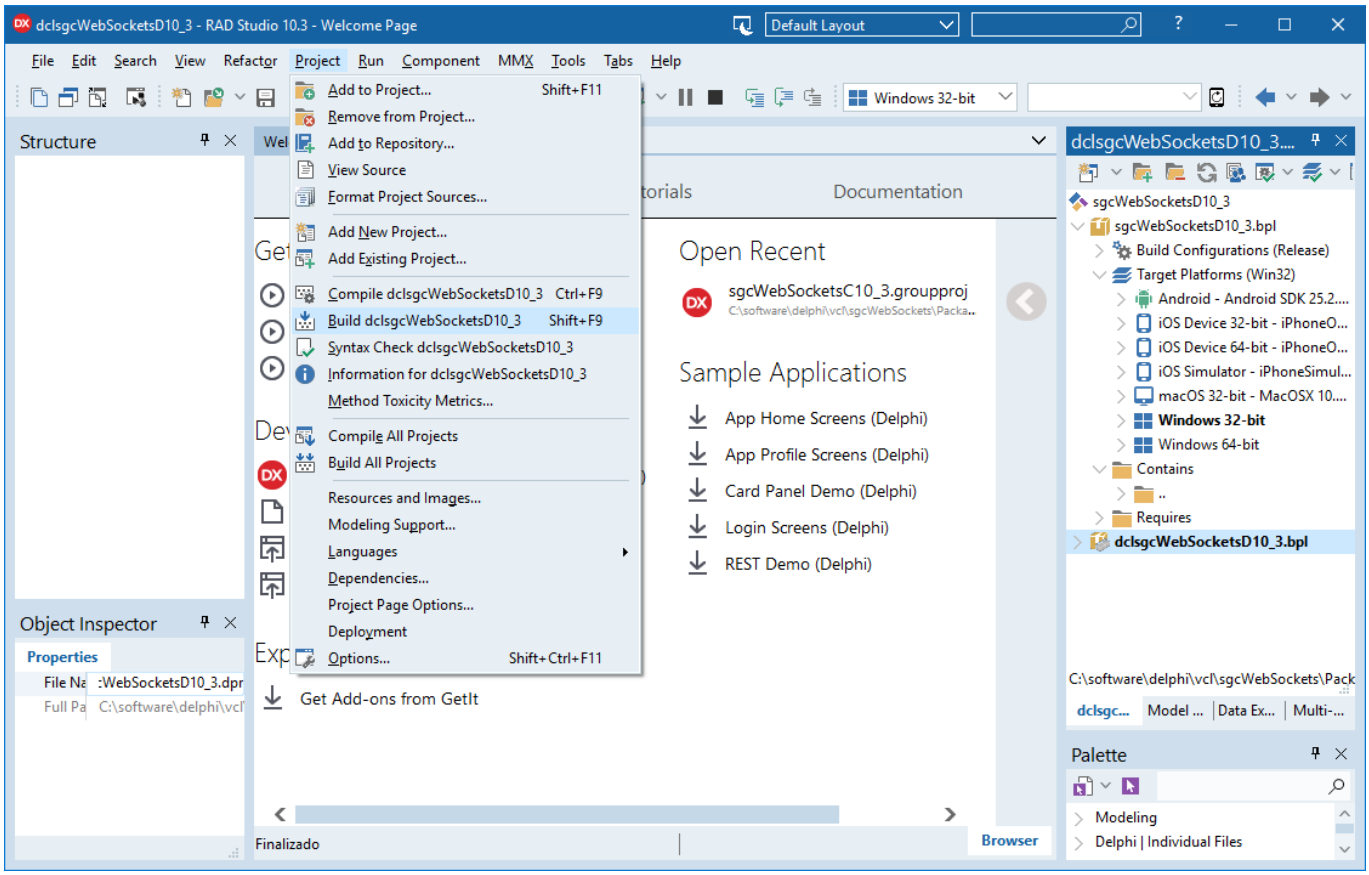
8. Select **MacOS 32** as Target Platform and build package.

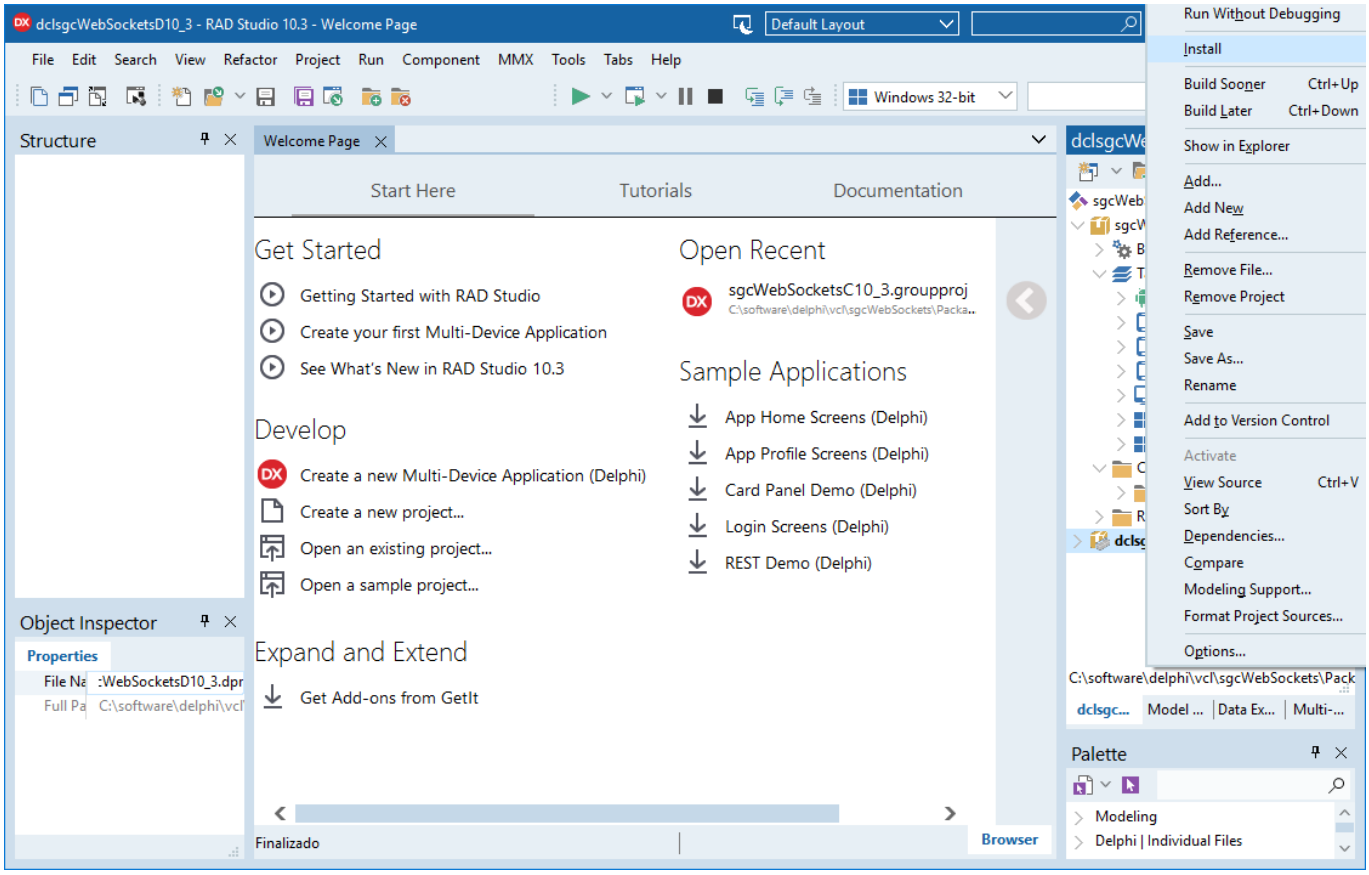


9. Select **Win32** as Target Platform and build package.

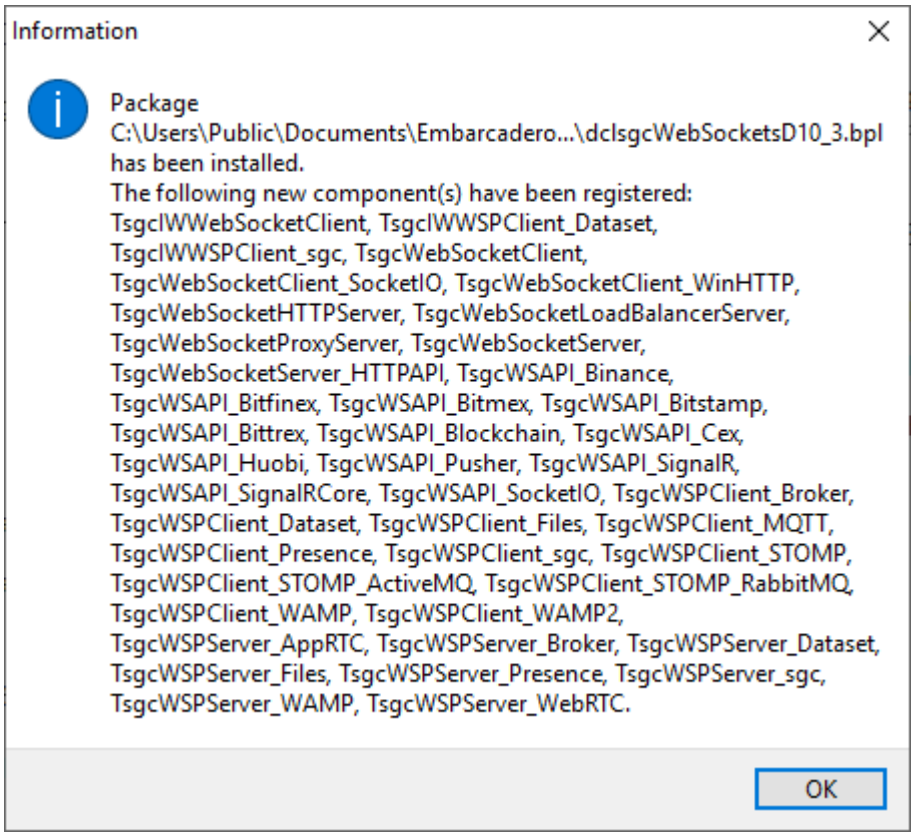


10. Once all runtime packages are compiled, select **design time package** (name starts with dcl) and first **build** and then **install** (design time packages only have Win32 as target platform).

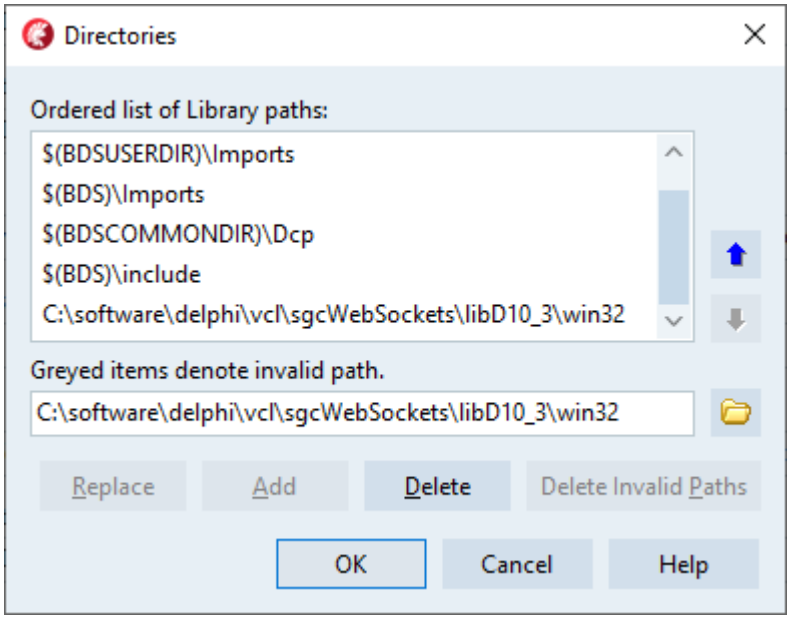




11. If installation is **successful** you will see a message with all components installed.



12. Then, you only need to add the directory where the compiled files are located to your **Rad Studio Library Path**. You must add this for every Target Platform (win32, win64, osx64...)



* If you are using the **Datasnap** servers, these are **NOT included** in the sgcWebSockets package because they cannot be installed; they are runtime-only components. In this case, you must add to your library path the Source folder too.

Install Errors

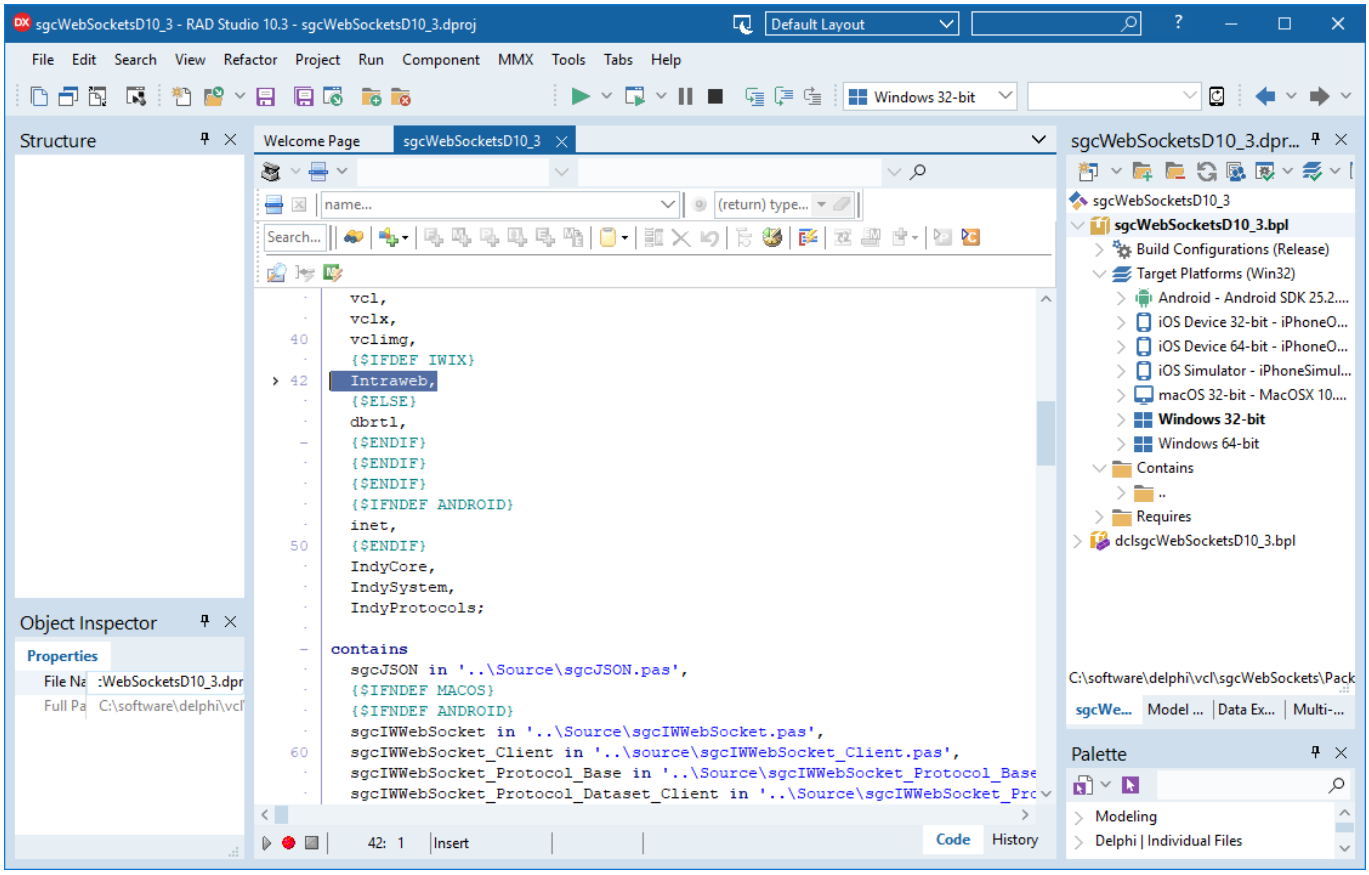
Sometimes you may get some errors installing components.

Intraweb package not found

sgcWebSockets is compiled using the **default Intraweb version** provided with Delphi. If you don't have Intraweb installed, you can **modify sgcVer.inc** file (located in Source folder).
Search your Delphi version and **comment all compiler defines for Intraweb** (starts with IW). **Example:** for Delphi 10.4 comment all compiler defines for Intraweb

```
{$IFDEF VER340} { Delphi  10.4 }
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
  {$DEFINE D2010}
  {$DEFINE DXE}
  {$DEFINE DXE2}
  {$DEFINE DXE3}
  {$DEFINE DXE4}
  {$DEFINE DXE5}
  {$DEFINE DXE6}
  {$DEFINE DXE7}
  {$DEFINE DXE8}
  {$DEFINE D10}
  {$DEFINE D10_1}
  {$DEFINE D10_2}
  {$DEFINE D10_3}
  {$DEFINE D10_4}
  {$DEFINE INDY10_1}
  {$DEFINE INDY10_2}
  {$DEFINE INDY10_5_5}
  {$DEFINE INDY10_5_7}
  {$DEFINE INDY10_5_8}
  {$DEFINE INDY10_6}
  {$DEFINE INDY10_6_0_5122}
  {$DEFINE INDY10_6_0_5169}
  {$DEFINE INDY10_6_2_5263}
  {$DEFINE INDY10_6_2_5366}
  {$DEFINE INDY10_6_2_D10_4}

  {$IFNDEF BCB}
    {$IFNDEF MACOS}
    {$IFNDEF ANDROID}
      {.$DEFINE IWIX}
      {.$DEFINE IWXI}
      {.$DEFINE IWXIV}
      {.$DEFINE IWXV}
    {$ENDIF}
    {$ENDIF}
    {$IFNDEF NEXTGEN}
      {$DEFINE SGC_JSON_INTF}
    {$ENDIF}
  {$ENDIF}
{$ENDIF}
```

If Intraweb is installed but it's a different version from the default that comes with Delphi, maybe your Intraweb package has a different name. Then open sgcWebSockets runtime package and **change Intraweb name** in project source.



Indy Package not found

sgcWebSockets requires Indy to install components in your IDE. Trial installation is compiled against Indy library provided with Delphi / CBuilder, so if you get a message like this:

[DCC Fatal Error] dclsgcWebSocketsDX.dpk(31): E2202 Required package 'IndyCore' not found

Most probably, you have a **newer Indy version**, so in order to install trial you must delete this version and install built-in indy version using Delphi / CBuilder setup.

If you have the full source code, then you only need to check:

1. Required Indy packages: IndyCore, IndySystem and IndyProtocols. If you have a newer Indy version, the packages most probably have a different name (including version), so access the menu **"Component / Install Packages"** and check which name have Indy packages and change accordingly in the project.

2. sgcWebSockets supports several Indy versions, there are compiler defines to allow compile for every Indy version. Open **sgcVer.inc**, located in the source folder, and change accordingly for your Indy version (is gsIdVersion of IdVers.inc Indy file). Some compiler defines:

```
{$DEFINE INDY10_1}
{$DEFINE INDY10_2}
{$DEFINE INDY10_5_5}
{$DEFINE INDY10_5_7}
{$DEFINE INDY10_5_8}
{$DEFINE INDY10_6}
{$DEFINE INDY10_6_0_5122}
{$DEFINE INDY10_6_0_5169}
{$DEFINE INDY10_6_2_5263}
{$DEFINE INDY10_6_2_5366}
{$DEFINE INDY10_6_2_D10_4}
```

c00000005 ACCESS_VIOLATION in CBuilder

If you compile a project using CBuilder and you get this error, set the following options in your project:

Project > Options > C++ Linker
uncheck "Link with Dynamic RTL"

Project > Options > Packages > Runtime Packages
uncheck "Link with runtime packages"

Unable to find package import: sgcWebSocketsCXXX.bpi in CBuilder Win64

When you compile runtime package for win64, you must compile Release and Debug.

Ambiguous reference System.ZLib.hpp and IdZLib.hpp CBuilder

sgcWebSockets Standard and Professional uses Indy for some components and Indy doesn't make use of ZLib unit, uses its own copy of ZLib: IdZLib, IdZLibHeaders... the project is linking to ZLib and indy ZLib units, so when compile, compiler doesn't know which is the correct reference because names are the same. There are 2 solutions:

1. Search where is included a link to System.ZLib.hpp and delete or move after IdZLibHeaders.hpp
2. Use the following conditional defines NO_USING_NAMESPACE_SYSTEM_ZLIB or DELPHIHEADER_NO_IMPLICIT_NAMESPACE_USE in your projects options to avoid the use of System.Zlib.hpp

Ambiguous reference System.ZLib.hpp and sgcIdZLib.hpp CBuilder

sgcWebSockets Enterprise uses a custom Indy version for some components and Indy doesn't make use of ZLib unit, uses its own copy of ZLib: sgcIdZLib, sgcIdZLibHeaders... the project is linking to ZLib and indy ZLib units, so when compile, compiler doesn't know which is the correct reference because names are the same. There are 2 solutions:

1. Search where is included a link to System.ZLib.hpp and delete or move after sgcIdZLibHeaders.hpp
2. Use the following conditional defines NO_USING_NAMESPACE_SYSTEM_ZLIB or DELPHIHEADER_NO_IMPLICIT_NAMESPACE_USE in your projects options to avoid the use of System.Zlib.hpp

Undefined reference to vTable for Sgcwebsocket... on CBuilder and Android

Use the following workaround to fix the error. Add the file libsgcwebsocketsC*.a which is located in the dcp/android default folder to your project using the menu "Project/ Add to Project".
Example: for CBuilder 11, add to your project the file "libsgcWebSocketsC11.a" which is located by default in the folder "C:\Users\Public\Documents\Embarcadero\Studio\22.0\DCP\Android\Release".

Checksum changed under Lazarus

This error can be raised while trying to install the components under Lazarus if the profile to build the IDE is not "Optimized IDE". The trial is compiled with the profile "Optimized IDE".

Cannot find X used by Y, incompatible ppu

Try the following workaround "Run / Clean up and rebuild" from the menu option.

Configure Install

In the source folder, there is a file called sgcVer.inc which includes all compiler defines for all Delphi, CBuilder and Lazarus IDEs.

Here you can customize your configuration for Intraweb, Indy... **usually there is no need to do any changes**, unless you want enable/disable some features.

Change carefully the compiler defines and contact us if you require assistance.

For every Delphi version, there is a section where you can configure all compiler defines, an example for Delphi 10.4

```
{$IFDEF VER340} { Delphi  10.4 }
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
  {$DEFINE D2010}
  {$DEFINE DXE}
  {$DEFINE DXE2}
  {$DEFINE DXE3}
  {$DEFINE DXE4}
  {$DEFINE DXE5}
  {$DEFINE DXE6}
  {$DEFINE DXE7}
  {$DEFINE DXE8}
  {$DEFINE D10}
  {$DEFINE D10_1}
  {$DEFINE D10_2}
  {$DEFINE D10_3}
  {$DEFINE D10_4}
  {$DEFINE INDY10_1}
  {$DEFINE INDY10_2}
  {$DEFINE INDY10_5_5}
  {$DEFINE INDY10_5_7}
  {$DEFINE INDY10_5_8}
  {$DEFINE INDY10_5_9}
  {$DEFINE INDY10_6}
  {$DEFINE INDY10_6_0_5122}
  {$DEFINE INDY10_6_0_5169}
  {$DEFINE INDY10_6_2_5263}
  {$DEFINE INDY10_6_2_5366}
  {$DEFINE INDY10_6_2_D10_4}

  {$IFNDEF BCB}
    {$IFNDEF MACOS}
    {$IFNDEF ANDROID}
      {.$DEFINE IWIX}
      {.$DEFINE IWXI}
      {.$DEFINE IWXIV}
      {.$DEFINE IWXV}
    {$ENDIF}
    {$ENDIF}
    {$IFNDEF NEXTGEN}
      {$DEFINE SGC_JSON_INTF}
    {$ENDIF}
  {$ENDIF}
{$ENDIF}
```

Indy

There are some compiler defines for Indy library. This depends on Indy version installed, by default is configured for Indy package included with Delphi. Indy version is gsIdVersion parameter of IdVers.inc Indy file.

Intraweb

If Intraweb is not installed, just comment out the compiler defines for Intraweb (those that start with IW...).

SGC_RESOURCES

By default sgcVer.inc declares `{$DEFINE SGC_RESOURCES}`, which causes the runtime packages to embed the resource file `sgcResources.RES`. This adds extra bytes to every executable that links statically against the sgcWebSockets source.
The resource contains the bundled JavaScript client code that an sgcWebSockets server can deliver over HTTP to a remote browser (so the browser can speak the sgc subprotocol). It is only needed when:

- You use an sgcWebSockets HTTP/WebSocket server that delivers the JS client bundle to browsers (typical for `TsgcWSProtocol_JS_*`-based servers).

It is **not** needed for:

- Delphi / C++Builder WebSocket clients (the data is consumed in native code, not in a browser).
- Servers that don't serve the bundled JS client (e.g. WebSocket-only servers, native-to-native servers).

To exclude the embedded resource, comment out the line in sgcVer.inc:

```
{.$DEFINE SGC_RESOURCES} { RESOURCES }
```

Alternatively, when running the source-edition Install Setup, uncheck the **Include Resources** option on the Options page; the installer will rewrite sgcVer.inc before compiling the packages.

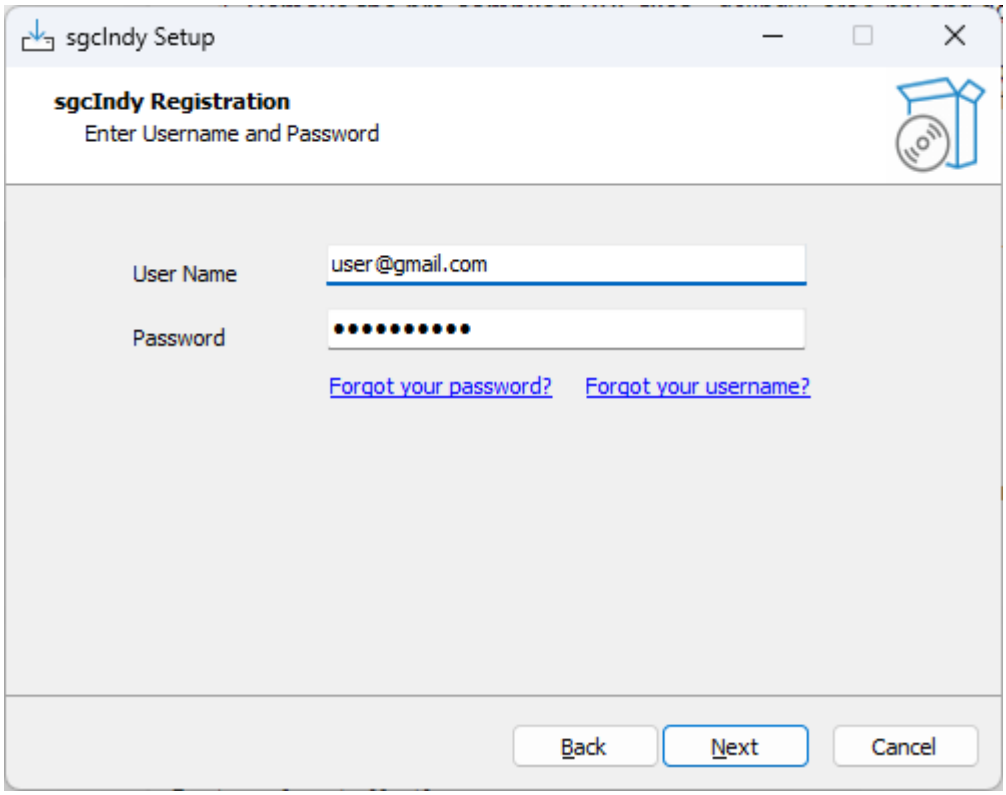
Install sgcIndy Package

Setup Installation

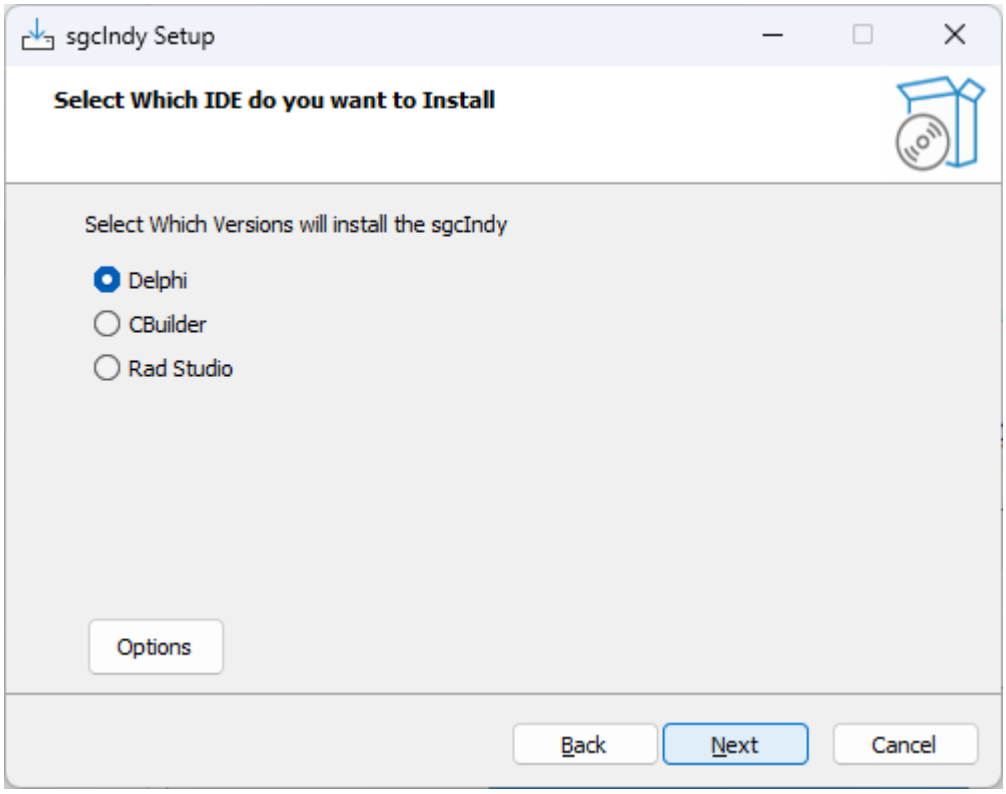
*Requires Windows Vista as minimum (Windows 2000, XP and Server 2003 are not supported).

Users who have purchased a license can install the sgcIndy package using the setup. Find below the step-by-step instructions for installing the package.

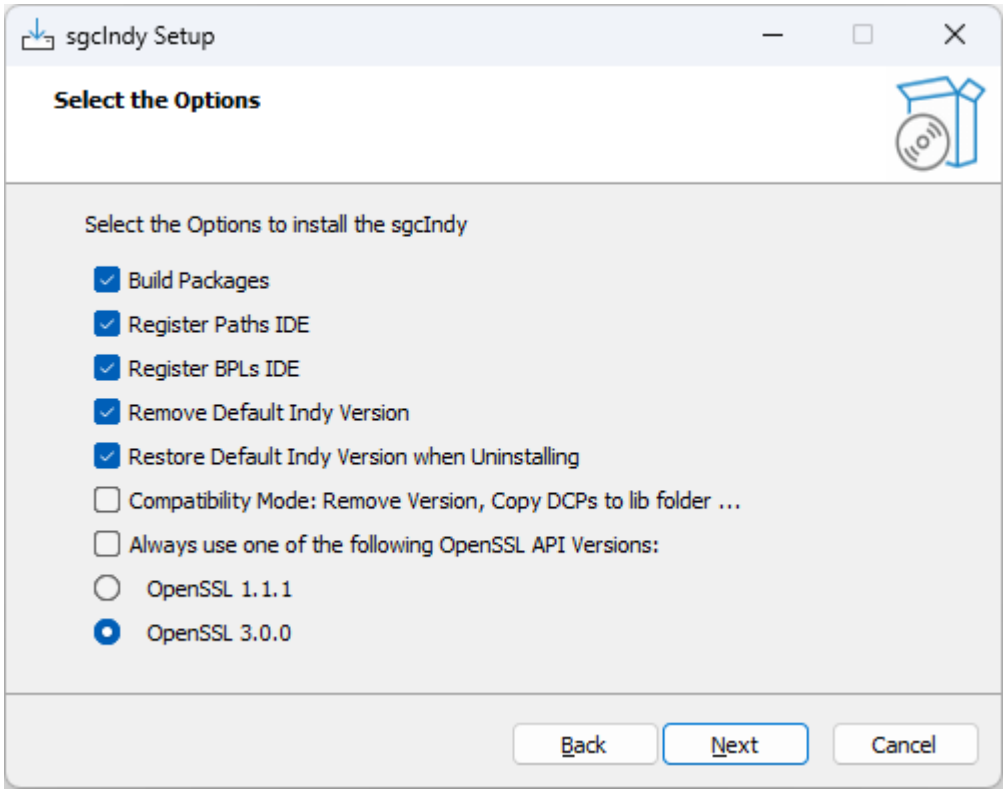
- Execute the Installer.
- First you must set your username/password of your private eSeGeCe account. This only must be entered one time, the next time you use the setup, the installer will read the latest value.



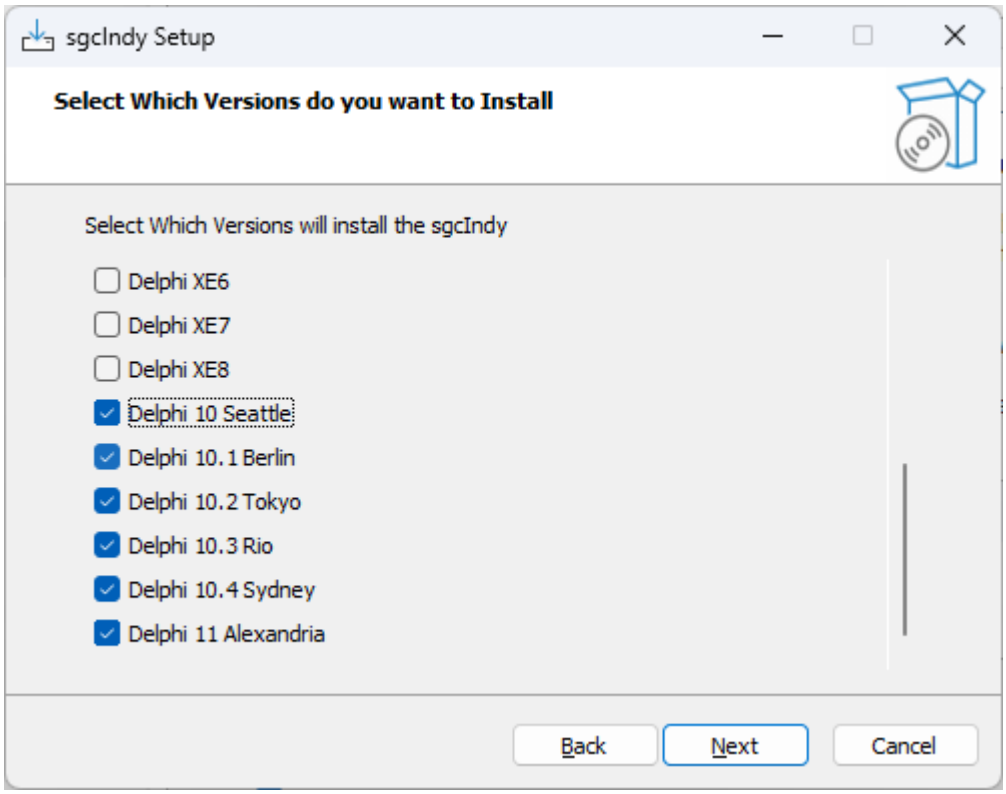
- If the user has logged in successfully, select whether you want to install in Delphi, CBuilder or Rad Studio IDE.



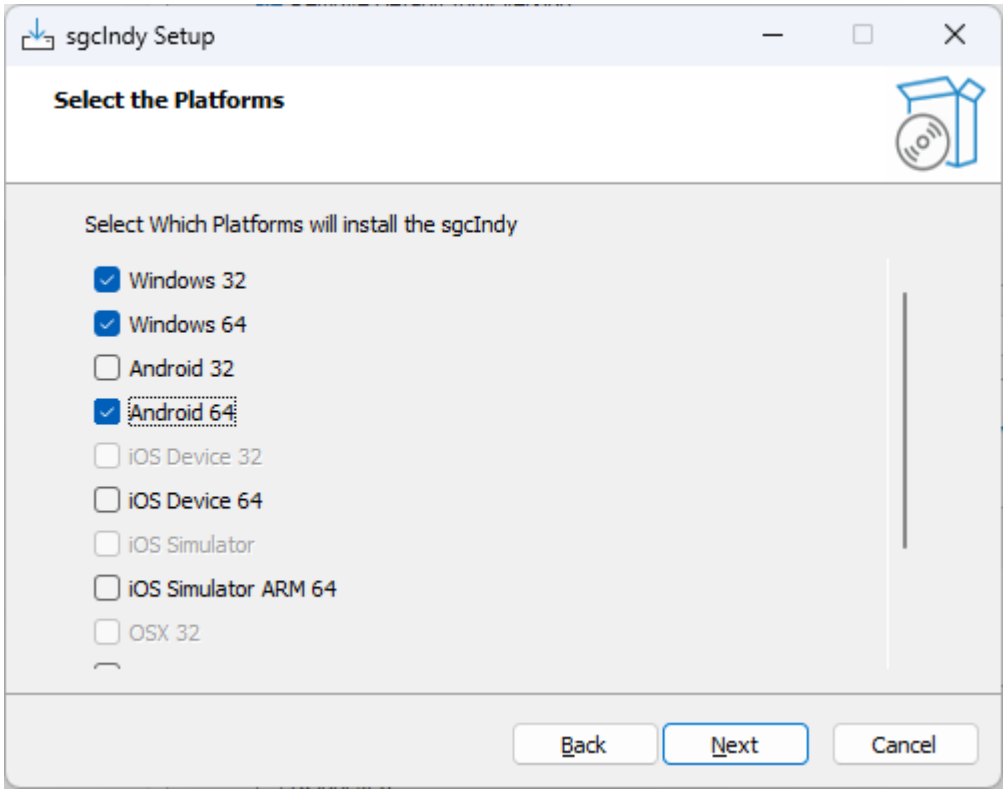
- There are some options that can be customized every time you use the installer, press the button **Options** to access these properties.
  - **Build Packages:** if selected, the installer will try to build the packages.
  - **Register Paths IDE:** if selected, the installer will register the required library paths in the IDE.
  - **Register BPLs IDE:** if selected and the installer has built the packages successfully, the installer will register the design-time package in the IDE.
  - **Remove Default Indy Version:** if selected, the installer will uninstall first the Standard Indy version that comes with Rad Studio.
  - **Restore Default Indy Version when Uninstalling:** if selected, the installer rollback the uninstalled Standard Indy version when the package is uninstalled.
  - **Compatibility Mode:** if selected, the dcp files are compiled without version and are copied to the Embarcadero/lib folder. Check this option if other packages are making use of Indy packages, like DevExpress. Additionally, it compiles the Embarcadero IP Abstraction units with the sgcIndy version installed.
  - **Always use of the following OpenSSL API Versions:** check this option if you want to force the use of OpenSSL 1.1.1 or OpenSSL 3.0.0 APIs



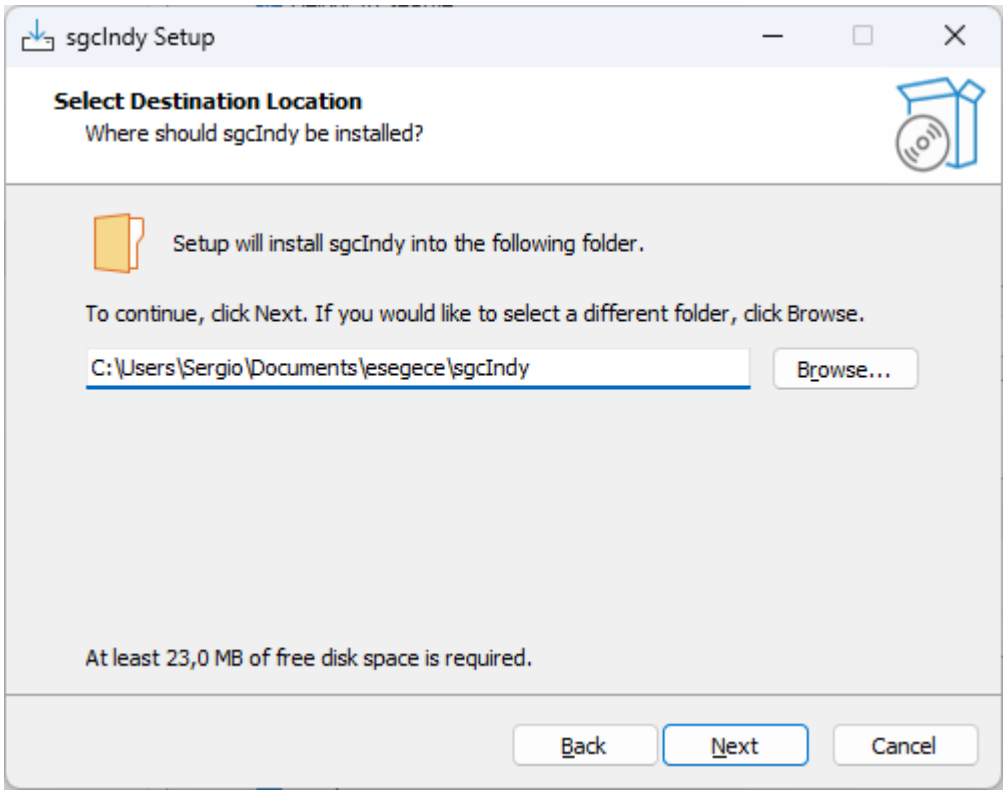
- Now you can select which IDE Versions you want to install. Only those IDE versions that the installer detect as installed, will be available.



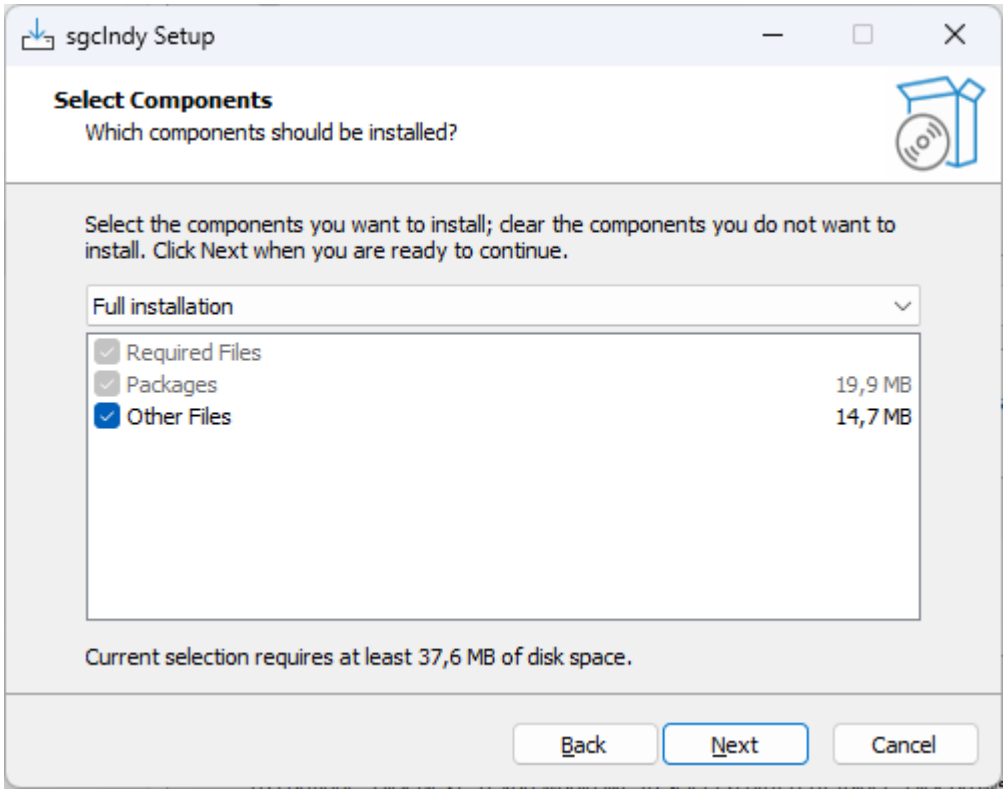
- Next step is select the Platforms.



- Select the folder where the package will be installed. If you reinstall the package, the installer will select by default the same folder selected in the previous install.



- Select which components to install.



- Finally, it will extract the files, compile and install the package and register the required paths in the IDE.

Install Errors

- MsBuild raises an error if the Length of the **Library Path is too high**, to fix this issue, try to delete unused paths from the library path. MsBuild has a limitation of 32K characters.

Manual installation

If Indy is **already installed**, it needs to be **uninstalled** first.

1. **Remove the pre-compiled BPL files** - dclIndyCoreX.bpl and dclIndyProtocolsX.bpl - from the IDE via the "Components > Install Packages" dialog.
2. Then **delete all of the existing binaries** (IndySystemX., IndyCoreX., IndyProtocolsX., dclIndyCoreX., and dclIndyProtocolsX.*) as well as delete any Indy 10 source files, if present.
3. Be sure to **check for files** in the IDE's \bin, \lib, and \source folders, \Indy subfolders, and OS system folders."

To **build the sgcIndy** package, you can either

1. **(Only CBuilder)** Use the command-line **FULLC#.BAT** script that corresponds to your CBuilder version.
2. **Open the individual DPK** files in the IDE and **compile** them, in the following order:
    1. IndySystemX.dpk (in Lib\System)
    2. IndyCoreX.dpk (in Lib\Core)
    3. IndyProtocolsX.dpk (in Lib\Protocols)
    4. dclIndyCoreX.dpk (in Lib\Core)
    5. dclIndyProtocolsX.dpk (in Lib\Protocols)

Once the Indy packages have been built, go to the menu **Components / Install Packages** and install the Indy Design-Time Packages

1. dclIndyCore*.bpl

2. dclIndyProtocols*.bpl

Finally set the paths in your IDE to the sgcIndy Packages.

Configure ZLib

ZLib version: 1.2.12

sgcWebSockets uses the ZLib compression when WebSocket Compression PerMessage Deflate Extension is enabled. By default, ZLib is statically linked with your application so there is no need to deploy the ZLib library.

If you want to use a specific library, add the following Conditional Define to your project:

SGC_DYNAMICLOAD_ZLIB

As an alternative, you can edit the file sgcIndy.inc (located in the source folder) and add the following line

{$DEFINE SGC_DYNAMICLOAD_ZLIB}

Finally, you must set the location where is the ZLib library, to do this, use the following method and pass the Full Path (without the name of the library) where is located

sgcIdZLibHeaders.IdZLibSetLibPath('c:\software\zlib');

**This configuration is only valid for sgcWebSockets Enterprise Edition with Source code.*

QuickStart

AI coding assistants

If you write sgcWebSockets code with an AI coding assistant, install the eSeGeCe agent skills first. They give the assistant the public API of the library, so it uses the components, properties and units that actually exist instead of inventing them. They are free, MIT licensed, and regenerated from the source at every release.

```
/plugin marketplace add esegece-com/agent-skills
/plugin install sgcwebsockets-delphi@esegece
```

Those two commands are for Claude Code and GitHub Copilot CLI. Cursor, Codex CLI, JetBrains Junie, Visual Studio and VS Code are covered at www.esegece.com/ai-agent-skills.

WebSockets Components

Creating a new WebSocket Server or WebSocket client is very simple, just create a new instance of the class, configure the Host / Port and set the property Active = true to start the process.

QuickStart WebSockets

HTTP Components

The HTTP/2 protocol allows you to create much faster HTTP Servers / Clients than using HTTP/1 protocol. The HTTP/2 Server is included in the WebSocket server while the HTTP/2 client is a dedicated component that implements the HTTP/2 protocol.

QuickStart HTTP

Threading Flow

sgcWebSockets components are threaded, which means that **connections run in secondary threads**. By **default**, the main **events are dispatched on the main thread**, this is useful when the number of events to dispatch is low, but for **better performance** you can configure the components where the **events are dispatched in the context of connection thread**. Read the following article which explains how to configure the threading flow:

How to Configure NotifyEvents

How to Build Applications

Building applications with the sgcWebSockets library is very easy. Just follow the next tips, which will help you **successfully build your application**.

Build

Fast Performance Server

sgcWebSockets has **2 server implementations**: 1 based on **Indy server** and another based on **HTTP.SYS Microsoft Server**. The latter is recommended for high-performance servers that need to handle thousands of concurrent connections. Check the following article which explains how to improve server performance.

Fast Performance Server

Memory Manager

Choosing an adequate memory manager can improve the performance of your application. Check the following article which shows a comparison between several memory managers.

Memory Manager

OpenSSL

When your application requires secure connections, usually **openSSL libraries** are required to **encrypt communications**, follow the next steps to configure successfully your application with openSSL libraries.

Configure OpenSSL

Indy

The Indy library is used as a base in some sgcWebSockets components, sgcWebSockets Enterprise edition includes a custom indy version which allows you to use openSSL 1.1.1 and openSSL 3.0.0, ALPN...

Indy

Linux (Lazarus)

If you compile a Lazarus project for Linux and you get this message:

```
Semaphore init failed (possibly too many concurrent threads)
```

Just add **cthreads** unit to your project file.

QuickStart | WebSockets

Let's start with a basic example where we need to create a Server WebSocket and 2 client WebSocket types: Application Client and Web Browser Client.

WebSocket Server

1. Create a new Window Forms Application
2. Drop a TsgcWebSocketServer onto a Form.
3. On Events Tab, Double click OnMessage Event, and type following code:

```
void OnMessage(TsgcWSConnection *Connection, const string Text)
{
  ShowMessage("Message Received From Client: " + Text);
}
```

4. Drop a Button onto the Form, Double Click and type this code:

```
TsgcWebSocketServer1->Active = true;
```

WebSocket Client

1. Create a new Window Forms Application
2. Drop a TsgcWebSocketClient onto a Form and configure Host and Port Properties to connect to Server.
3. Drop a TButton in a Form, Double Click and type this code:

```
TsgcWebSocketClient1->Active = true;
```

4. Drop a Button onto the Form, Double Click and type this code:

```
TsgcWebSocketClient1->WriteData("Hello Server From VCL Client");
```

Web Browser Client

1. Create a new HTML file

2. Open file with a text editor and copy following code:

```
<html>
<head>
<script type="text/javascript" src="http://host:port/sgcWebSockets.js"></script>
</head>
<body>
<a href="javascript:var socket = new sgcWebSocket('ws://host:port');">Open</a>
<a href="javascript:socket.send('Hello Server From Web Browser');">Send</a>
</body>
</html>
```

You need to replace host and port in this file for your custom Host and Port!!

3. Save File and that's all, you have configured a basic WebSocket Web Browser Client.

How To Use

1. Start Server Application and press button to start WebSocket Server to listen new connections.

2. Start Client Application and press button1 to connect to server and press button2 to send a message. On Server Side, you will see a message with text sent by Client.

3. Open then HTML file with your Web Browser (Chrome, Firefox, Safari or Internet Explorer 10+), press Open to open a connection and press send, to send a message to the server. On Server Side, you will see a message with a text sent by Web Browser Client.

Linux Compiler

Simple Server example (listening on port 5000).

```
program sgcWebSockets_linux;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils, sgcWebSocket, sgcWebSocket_Types;

var
  oServer: TsgcWebSocketServer;

begin
  try
    oServer := TsgcWebSocketServer.Create(nil);
    oServer.Port := 5000;
    oServer.NotifyEvents := neNoSync;
    oServer.Active := True;

    while oServer.Active do
      Sleep(10);
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

Linux (Lazarus)

If you compile a Lazarus project for Linux and you get this message:

```
Semaphore init failed (possibly too many concurrent threads)
```

Just add **cthreads** unit to your project file.

QuickStart | HTTP

Let's start with a basic example where we need to create a HTTP/2 Server and a HTTP/2 client.

HTTP/2 Server

1. Create a new Window Forms Application
2. Drop a TsgcWebSocketHTTPServer onto a Form.
3. On Events Tab, Double click OnCommandGet Event, and type following code:

```
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Document == "/")
  {
    AResponseInfo->ContentText = "<html><head><title>Test Page</title></head><body></body></html>";
    AResponseInfo->ContentType = "text/html";
    AResponseInfo->ResponseNo = 200;
  }
}
```

4. By default, the server only enables HTTP/1 connections, so enable HTTP/2 options in the property **HTTP2Options.Enabled = true**, and then configure the SSL Options. Secure connections require OpenSSL libraries.

```
TsgcWebSocketHTTPServer1->Port = 443;
TsgcWebSocketHTTPServer1->SSL = true;
TsgcWebSocketHTTPServer1->SSLOptions->CertFile = "server cert file";
TsgcWebSocketHTTPServer1->SSLOptions->KeyFile = "server private key file";
TsgcWebSocketHTTPServer1->SSLOptions->RootCertFile = "server root cert file";
TsgcWebSocketHTTPServer1->SSLOptions->OpenSSL_Options->APIVersion = oslAPI_1_1;
TsgcWebSocketHTTPServer1->SSLOptions->Port = 443;
TsgcWebSocketHTTPServer1->SSLOptions->Version = tls1_3;
```

5. Drop a Button onto the Form, Double Click and type this code:

```
TsgcWebSocketHTTPServer1->Active = true;
```

HTTP/1 Client

1. Create a new Window Forms Application
2. Drop a TButton in a Form, Double Click and type this code:

```
TsgcHTTP1Client *oHTTP1 = new TsgcHTTP1Client();
try
{
  ShowMessage(oHTTP1->Get("https://127.0.0.1"));
}
__finally
{
  oHTTP1->Free();
}
```

HTTP/2 Client

1. Create a new Window Forms Application
2. Drop a TButton in a Form, Double Click and type this code:

```
TsgcHTTP2Client *oHTTP2 = new TsgcHTTP2Client();
try
{
  ShowMessage(oHTTP2->Get("https://127.0.0.1"));
}
__finally
{
  oHTTP2->Free();
}
```

QuickStart | Threading Flow

sgcWebSockets components are threaded, for example, **TsgcWebSocketHTTPServer** (based on Indy library) creates one thread for every connection while **TsgcWebSocketServer_HTTPAPI** (based on Microsoft HTTP.SYS) runs a pool of threads and the connections are handled by this pool of threads (max of 64 threads) and **TsgcWebSocketClient** runs its own thread to asynchronously process the responses from the WebSocket server.

By default, there is a property called **NotifyEvents**, which has the value neAsynchronous. This means that when a WebSocket client receives a message, this message is queued and is dispatched on the main thread by OS later. This works well for clients that do not receive a lot of messages and for ease of use, because it does not require synchronizing with the main thread when you want, for example, to update a control on your form.

But when the server / client must process several messages in short period of time, it is better to change this threading flow to one where the events are dispatched in the context of the connection thread. To do this, just set NotifyEvents property to **neNoSync**, this way, when for example a client receives a message from server, this message will be dispatched in the context of a secondary thread, so if you need to update a control of your form, first synchronize with the main thread and then update the form control (because form controls are not thread safe). The same applies if you want access to a shared object, you need to implement your own synchronization methods.

Threading Flow Easy Mode (NotifyEvents = neAsynchronous) and Low Performance

This is the threading flow by default and it's usually used on demo samples. Select this mode if you do not expect to handle several messages per second and you need to update form controls or access shared objects.

NotifyEvents = neAsynchronous

Threading Flow Best Performance (NotifyEvents = neNoSync)

Set this threading flow for server components and for clients that need high performance because you expect to handle several messages. Using this configuration, the events are dispatched in the context of connection thread, so in order to update a Form control, first synchronize with the main thread.

NotifyEvents = neNoSync

How Synchronize Main Thread

You can synchronize with Main Thread calling TThread.Synchronize or TThread.Queue, both methods can be used and selecting one or the other depends on how you want to implement synchronization.

TThread.Synchronize

This method is blocking, which means that when you call Synchronize, the code blocks until it synchronizes with the main thread.

TThread.Queue

This method is non blocking, so when you call queue, the message is queued and will be dispatched later.

Example Code

Update a Memo with the messages received from WebSocket Client.

```
void OnClientMessage(TsgcWSConnection *Connection, string aText)
{
```

```
  TThread::Queue(NULL,
    [&]()         {
          memo1->lines->add(aText);
        });
}
```

QuickStart | Build

Building an application with the sgcWebSockets library is very easy. Just keep in mind whether your components require OpenSSL libraries or not. If your applications require secure connections, openSSL libraries must be deployed (except if you use SChannel for windows on Client Components).

For **Windows applications**, it is enough to deploy the OpenSSL libraries in the same folder where the application is located.

For other personalities check the following articles:

- Build OSX Application
- Build Android Application
- Build iOS Application

CBuilder DEBUG

If you are using CBuilder and want to debug the sgcWebSockets library, follow the next steps:

- Go into the Project Options.
- Enable Use debug .dcus under Delphi Compiler > Compiling.
- Disable Link with Dynamic RTL under C++ Linker.
- Disable Link with Runtime Packages under Packages > Runtime Packages.

You will then be able to step into the VCL/RTL source code.

Build | OSX Application

In order to build an OSX Application with sgcWebSockets library you must follow the steps from Embarcadero website to build an OSX Application.

Install PASServer in MacOS

http://docwiki.embarcadero.com/RADStudio/Sydney/en/PAServer,_the_Platform_Assistant_Server_Application

Obtain a Developer ID Certificate

Login with a valid Apple Developer Account to https://developer.apple.com and create a new "Developer ID Application" from Certificates menu.

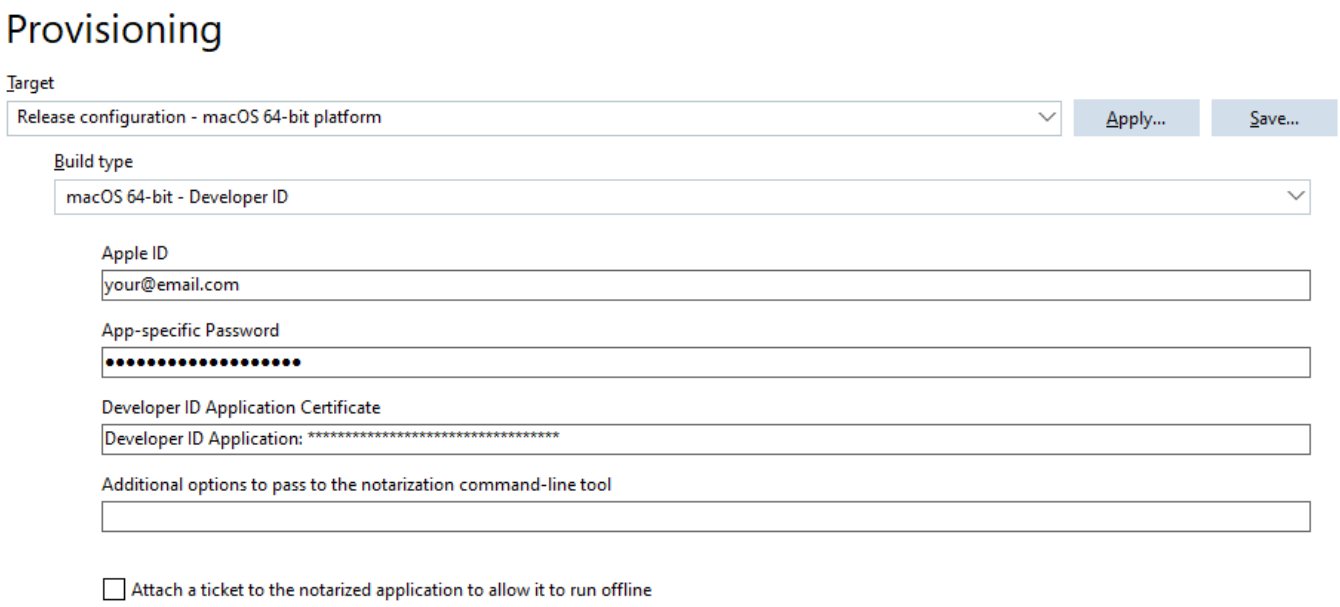
http://docwiki.embarcadero.com/RADStudio/Rio/en/MacOS_Notarization

Create a new Apple Id

Then go to https://appleid.apple.com/account to create a new Apple Id

Configure Provisioning

Finally, open the menu **Project / Options / Provisioning** and fill the required data to notarize an OSX Application.



If your project requires some libraries, don't forget to include in the menu **Project / Deployment**. Set Remote Path to "Contents\MacOS\"

Local Path	Local Name	Type	Configurat...	Platforms	Remote Path	Remote Name	Remote Status	Overwrite
☑	libcrypto.1.0.0.dylib	File	Release	[OSX64]	Contents\MacOS\	libcrypto.1.0.0.dylib	Not Connected	Always
☑ OSX64\Release\	sgcClientMobile	ProjectOutput	Release	[OSX64]	Contents\MacOS\	sgcClientMobile	Not Connected	Always
☑ $(BDS)\bin\	delphi_PROJECTICNS.icns	ProjectOSXRes...	Release	[OSX64]	Contents\Resources\	sgcClientMobile.icns	Not Connected	Always
☑ OSX64\Release\	sgcClientMobile.dSYM	ProjectOSXDe...	Release	[OSX64]	..\$(PROJECTNAME).ap...	sgcClientMobile	Not Connected	Always
☑ OSX64\Release\	sgcClientMobile.info.plist	ProjectOSXInf...	Release	[OSX64]	Contents\	Info.plist	Not Connected	Always
☑	libssl.1.0.0.dylib	File	Release	[OSX64]	Contents\MacOS\	libssl.1.0.0.dylib	Not Connected	Always
☑ OSX64\Release\	sgcClientMobile.entitle...	ProjectOSXEnt...	Release	[OSX64]	..\	sgcClientMobile.entitle...	Not Connected	Always

These libraries will be automatically signed when the application is notarized, you can check if the library has been signed using the following command:

```
codesign -dv --verbose=4 libcrypto.1.1.dylib
```

Read more about How Configure openSSL OSX.

Build | Android Application

In order to build an Android Application with the sgcWebSockets library, you must follow the steps from the Embarcadero website to build an Android Application.

Creating an Android App

http://docwiki.embarcadero.com/RADStudio/Sydney/en/Creating_an_Android_App

Project Deployment

If your project requires some libraries, don't forget to include in the menu **Project / Deployment**. Set Remote Path to ".\assets\internal"

Local Path	Local Name	Type	Configurat...	Platforms	Remote Path	Remote Name
$(BDS)\bin\Artwork\...	FM_LauncherIcon_72x7...	Android_Laun...	Release	[Android64]	res\drawable-hdpi\	ic_launcher.png
$(BDS)\bin\Artwork\...	FM_SplashImage_960x7...	Android_Splas...	Release	[Android64]	res\drawable-xlarge\	splash_image.png
..\..\..\apps\Third-pa...	libssl.so	File	Release	[Android64]	.\assets\internal	libssl.so
$(BDS)\lib\android\...	libnative-activity.so	AndroidLibnat...	Release	[Android64]	library\lib\armeabi\	libsgcClientMobile.so
Android64\Release\	strings.xml	Android_Strings	Release	[Android64]	res\values\	strings.xml
$(BDS)\bin\Artwork\...	FM_SplashImage_640x4...	Android_Splas...	Release	[Android64]	res\drawable-large\	splash_image.png
Android64\Release\	classes.dex	AndroidClasse...	Release	[Android64]	classes\	classes.dex
Android64\Release\	styles-v21.xml	AndroidSplash...	Release	[Android64]	res\values-v21\	styles.xml
$(BDS)\bin\Artwork\...	FM_LauncherIcon_48x4...	Android_Laun...	Release	[Android64]	res\drawable-mdpi\	ic_launcher.png
$(BDS)\bin\Artwork\...	FM_SplashImage_470x3...	Android_Splas...	Release	[Android64]	res\drawable-normal\	splash_image.png
$(BDS)\bin\Artwork\...	FM_SplashImage_426x3...	Android_Splas...	Release	[Android64]	res\drawable-small\	splash_image.png
$(BDS)\lib\android\...	libnative-activity.so	AndroidLibnat...	Release	[Android64]	library\lib\armeabi-v7a\	libsgcClientMobile.so
Android64\Release\	libsgcClientMobile.so	ProjectOutput	Release	[Android64]	library\lib\arm64-v8a\	libsgcClientMobile.so
..\..\..\apps\Third-pa...	libcrypto.so	File	Release	[Android64]	.\assets\internal	libcrypto.so
Android64\Release\	styles.xml	AndroidSplash...	Release	[Android64]	res\values\	styles.xml

Read more about How Configure openSSL Android.

Build | iOS Application

In order to build an iOS Application with the sgcWebSockets library, you must follow the steps from the Embarcadero website to build an iOS Application.

Install PASServer in MacOS

http://docwiki.embarcadero.com/RADStudio/Sydney/en/PAServer,_the_Platform_Assistant_Server_Application

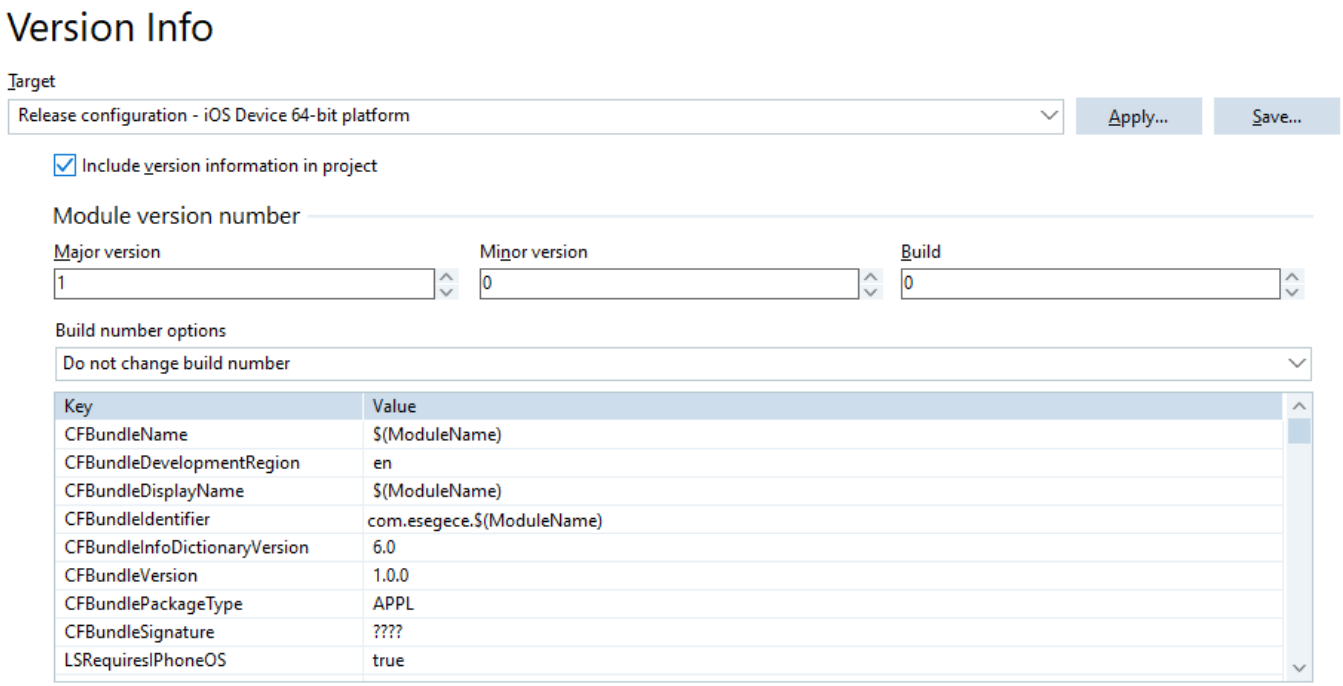
Obtain an iOS Development Certificate

Login with a valid Apple Developer Account to https://developer.apple.com and create a new "iOS Development Certificate" from Certificates menu.
Create a new Identifier for your iOS apps and a new provisioning profile.

http://docwiki.embarcadero.com/RADStudio/Sydney/en/IOS_Mobile_Application_Development

Configure Bundle Identifier

Open the menu **Project / Options / Application / Version Info** and set your Bundle Identifier



Deployment

If your project requires some static libraries, copy these libraries in the Embarcadero lib/iosDevice64 folder:

- C:\Program Files (x86)\Embarcadero\Studio\<IDE Version>\lib\iosDevice64\debug
- C:\Program Files (x86)\Embarcadero\Studio\<IDE Version>\lib\iosDevice64\release

Read more about How Configure openSSL iOS

Provisioning

Finally, check in the menu **Project / Options / Deployment**, if the certificate has been successfully loaded.

Provisioning

Target

Release configuration - iOS Device 64-bit platform | Apply... | Save...

<Use environment options (Auto)>

Developer Certificate:

<Auto> - To select a certificate, first select a valid provision profile

Provision Profile

Name:
provisioning esegece
File Path:
/Users/sergio/Library/MobileDevice/Provisioning Profiles/
Application Identifier:

Developer program Name:
Sergio Gomez

Developer Certificate

iPhone Developer: Sergio Gomez

Current Bundle Identifier:
com.esegece.sgcClientMobile

Fast Performance Servers

Servers based on Indy Library

TsgcWebSocketServer and TsgcWebSocketHTTPServer are based on Indy library, so every connection is handled by a thread, so if you have 1000 concurrent connections, you will have, at least, 1000 threads to handle these connections. When performance is important, you must do some "tweaks" to increase performance and improve server work. **From sgcWebSockets 4.3.3 Indy servers support IOCP too**, you can read more.

Use the following tips to increase server performance.

1. Set in Server component property **NotifyEvents := neNoSync**. This means that events are raised in the context of connection thread, so there is no synchronization mechanism. If you must access VCL controls or shared objects, use your own synchronization mechanisms.

2. Set in Server component property **Optimizations.Connections.Enabled := True**. If you plan to have more than 1000 concurrent connections in your server, and you call Server.WriteData method a lot, enable this property. Basically, it saves connections in a cache list where searches are faster than accessing to Indy connections list.

    2.1 CacheSize: is the number of connections stored in a fast cache. Default value = 100.
    2.2 GroupLevel: creates internally several lists split by the first character, so if you have lots of connections, searches are faster. Default value = 1.

3. Set in Server component property **Optimizations.Channels.Enabled := True**. Enabling this property, channels are saved in a list where searches are faster than previous method.

4. Set in Server component property **Optimizations.ConnectionsFree.Enabled := True**. If this property is enabled, every time there is a disconnection, instead of destroying TsgcWSConnection, the object is stored in a List and every X seconds, all objects stored in this list are destroyed. Enabling this property the memory consumption will be higher, so if you have a lot of disconnections in a short period of time, set this property to false.

    4.1 Interval: number of seconds where all disconnected connections stored in a list are destroyed. By default is 60.

5. By default, sgcWebSockets uses **Critical Sections** to protect access to shared objects. But you can use TMonitor or SpinLocks instead of critical sections. Just compile your project with one of the following compiler defines

    3.1 {$DEFINE SGC_SPINLOCKS}
    3.2 {$DEFINE SGC_TMONITOR}

6. Use latest **FastMM4**, you can download from: https://github.com/pleriche/FastMM4
    FastMM4 is a very good memory manager, but sometimes doesn't scale well with multi-threaded applications. Use the following compiler define in your application:

    {$DEFINE UseReleaseStack}

    Then, add FastMM4 as the first unit in your project uses and compile again. For a high concurrent server, you will note an increase in performance.
    This tweak does the following: If a block cannot be released immediately during a FreeMem call the block will be added to a list of blocks that will be freed later, either in the background cleanup thread or during the next call to FreeMem.

7. Better than FastMM4, use the latest **FastMM5**, you can download from: https://github.com/pleriche/FastMM5
    This is a new version from the same developer of FastMM4, supports Delphi XE3 and later and can be used on Windows32 and Windows64.
    FastMM5 is dual licensed, so there are 2 licenses: GPL and Commercial. So if you want to use it in commercial projects, you must purchase a license.
    Find below a grid which compares the performance between FastMM4 and FastMM5, doing 100.000 websocket requests and responses using 1, 10, 100, 500 and 1000 concurrent clients. The performance under FastMM5 is much better, in multithreaded applications, than using FastMM4.

Clients	Windows	FMM4	FMM5	Difference
1	Win32	4135	4214	1,91%
	Win64	4052	4520	11,55%
10	Win32	4214	1729	-58,97%
	Win64	4104	1875	-54,31%
100	Win32	3958	1604	-59,47%
	Win64	3958	1614	-59,22%
500	Win32	4098	1723	-57,96%
	Win64	5333	1791	-66,42%
1000	Win32	5927	2208	-62,75%
	Win64	8166	2229	-72,70%

Indy Server Windows

sgcWebSockets Enterprise Edition supports **IOCP** on Windows, this means that instead of creating 1 thread for every connection a pool of threads handle all the connections. To enable IOCP, just set the IOHandler to IOCP.

IOHandlerOptions.IOHandlerType = iohIOCP

The property IOHandlerOptions.IOCP allows you to customize the IOCP properties.

- **IOCPThreads:** these are the threads used to handle the connections, by default the value is zero which means the threads will be calculated automatically using the number of processors (for Delphi 7 to Delphi 2007 this value is set to 32 because the CPU count function is not supported).
- **WorkOpThreads:** set a value greater than zero if you want that the requests for every connection are handled always by the same thread. By default, IOCP requests are handled by random threads, if you want that the connections are handled by always the same thread, set a value greater than zero. Example: if you set WorkOpThreads = 32, the server will create 32 threads and every time there is a new request, if the connection was already processed previously it will be queued in the same thread.

IOCP is recommended when you want to handle thousands of concurrent connections.

Indy Server Linux

sgcWebSockets Enterprise Edition supports **EPOLL** on Linux, this means that instead of creating 1 thread for every connection a pool of threads handle all the connections. To enable EPOLL, just set the IOHandler to EPOLL.

IOHandlerOptions.IOHandlerType = iohEPOLL

The property IOHandlerOptions.EPOLL allows customizing the EPOLL properties.

- **EPOLLThreads:** these are the threads used to handle the connections, by default the value is zero which means the threads will be calculated automatically using the number of processors.

- **WorkOpThreads:** set a value greater than zero if you want that the requests for every connection are handled always by the same thread. By default, EPOLL requests are handled by random threads, if you want that the connections are handled by always the same thread, set a value greater than zero. Example: if you set WorkOpThreads = 32, the server will create 32 threads and every time there is a new request, if the connection was already processed previously it will be queued in the same thread.

EPOLL is recommended when you want to handle thousands of concurrent connections.

Server Based on HTTP.SYS

TsgcWebSocketServer_HTTPAPI component is based on Microsoft HTTP API and it's designed to work with IOCP, so it's recommended when the server must handle thousands of connections but it has the limitation that can only run on Windows.

The server can handle **WebSocket** and **HTTP/2** protocols on the same port and can work with other implementations because it can be configured to only handle some endpoints.
Example: you can configure this server to handle websocket connections with our sgcWebSockets library and let other implementations / third-parties or whatever use other endpoints.

- Endpoint: https://server/ws will handle connections that use WebSocket protocol using sgcWebSockets
- Endpoint: https://server/other will handle connections using another library.

Use latest **FastMM5**, you can download from: https://github.com/pleriche/FastMM5
This is a new version from the same developer of FastMM4, supports Delphi XE3 and later and can be used on Windows32 and Windows64.
FastMM5 is dual licensed, so there are 2 licenses: GPL and Commercial. So if you want to use it in commercial projects, you must purchase a license.
Find below a grid which compares the performance between FastMM4 and FastMM5, doing 100.000 websocket requests and responses using 1, 10, 100, 500 and 1000 concurrent clients. The performance under FastMM5 is much better, in multithreaded applications, than using FastMM4.

Clients	Windows	FMM4	FMM5	Difference
1	Win32	5364	5182	-3,39%
	Win64	5057	5026	-0,61%
10	Win32	4922	1744	-64,57%
	Win64	4958	1770	-64,30%
100	Win32	3359	1682	-49,93%
	Win64	3979	1536	-61,40%
500	Win32	2364	1890	-20,05%
	Win64	2901	1666	-42,57%
1000	Win32	3296	1968	-40,29%
	Win64	4469	1989	-55,49%

Memory Manager

Recently a new version of FastMM, developed by Pierre le Riche, has been released, the new version is called **FastMM5** and has been rewritten to improve the performance on multi threaded applications, can be configured for better speed or less memory usage and more.

Support from Delphi **XE3 Compiler** and can used on **Windows32** and **Windows64**.

FastMM5 is **dual licensed**, so there are 2 licenses: **GPL** and **Commercial**. So if you want use in commercial projects, you must purchase a license. More details here

https://github.com/pleriche/FastMM5

FastMM4 has a new fork, called **FastMM4-AVX**, developed by Maxim Masiutin, which adds very interesting features like: more efficient synchronization, AVX instructions for faster memory copy, speed improvements and more. FastMM4-AVX is dual licensed: MPL and GPL. More details here:

https://github.com/maximmasiutin/FastMM4-AVX

Configuration

In order to test the performance with our components, a new windows console application has been created, **sgcBenchmark** which will be used to measure the performance of every memory manager using our sgcWebSockets components.

The test is very simple, a client (or more than one client) connects to a server, sends a message to server and server replies with the same message to client. This is repeated 100.000 times. The tests are repeated changing the number of concurrent clients, first 1, then 10, 100... the measured time is the time elapsed between the first message sent by client and the last message received from server (so the time used to connect to server is not measured).

The benchmark will compare the performance using the Default Memory Manager that comes with Delphi 10.4.1, FastMM5 and FastMM4-AVX

Benchmark Indy WebSocket Server

In the first Benchmark, the Server used is the Indy WebSocket Server, this server is based on Indy TCP Server, so every connection creates 1 thread.

The values are measured in milliseconds, so for example, the first test that is done with 1 client in Windows32 platforms, using the default memory manager takes 4135 milliseconds, using FastMM5 takes 4214 milliseconds and using FastMM4-AVX takes 4823 milliseconds. The percentage calculated is against the reference value, in this case against the Default memory manager that comes with delphi, as much lower is the percentage, better performance has.

The Benchmark has been done 3 times and the values showed are the sum of the benchmarks / 3.

For the benchmark, the server used was:

- Windows 2016 Server Datacenter
- 16 Virtual Processors
- 32 GB RAM
- 2.2 GHz

The Delphi version used was Delphi 10.4.1, and the latest FastMM5 and FastMM4-AVX versions from github servers.

Find below the result of the benchmark.

Clients	Platform	Default (ms)	FMM5 (ms)	FMM5 (%)	FMM4-AVX (ms)	FMM4-AVX (%)
1	Win32	4135	4214	1.91%	4823	16.64%
1	Win64	4052	4520	11.55%	4328	6.81%
10	Win32	4214	1729	-58.97%	1828	-56.62%
10	Win64	4104	1875	-54.31%	1651	-59.77%
100	Win32	3958	1604	-59.47%	1583	-60.01%
100	Win64	3958	1614	-59.22%	1635	-58.69%
500	Win32	4098	1723	-57.96%	1854	-54.76%
500	Win64	5333	1791	-66.42%	1833	-65.63%
1000	Win32	5927	2208	-62.75%	2328	-60.72%
1000	Win64	8166	2229	-72.70%	2234	-72.64%

Benchmark HTTP.SYS Server

In the second Benchmark, the Server used is the HTTP.SYS WebSocket Server, this server is based on HTTP API Microsoft Framework and the connections are handled by a pool of threads.

The values are measured in milliseconds, so for example, the first test that is done with 1 client in Windows32 platforms, using the default memory manager takes 5364 milliseconds, using FastMM5 takes 5182 milliseconds and using FastMM4-AVX takes 5838 milliseconds. The percentage calculated is against the reference value, in this case against the Default memory manager that comes with Delphi, as much lower is the percentage, better performance has.

The Benchmark has been done 3 times and the values showed are the sum of the benchmarks / 3.

For the benchmark, the server used was:

- Windows 2016 Server Datacenter
- 16 Virtual Processors
- 32 GB RAM
- 2.2 GHz

The Delphi version used was Delphi 10.4.1, and the latest FastMM5 and FastMM4-AVX versions from github servers.

Find below the result of the benchmark.

Clients	Platform	Default (ms)	FMM5 (ms)	FMM5 (%)	FMM4-AVX (ms)	FMM4-AVX (%)
1	Win32	5364	5182	-3.39%	5838	8.84%
1	Win64	5507	5206	-0.61%	5135	1.54%
10	Win32	4922	1744	-64.57%	2088	-57.58%

10	Win64	4958	1770	-64.30%	1953	-60.61%
100	Win32	3359	1682	-49.93%	2244	-33.19%
100	Win64	3979	1536	-61.40%	1859	-53.28%
500	Win32	2364	1890	-20.05%	2344	-0.85%
500	Win64	2901	1666	-42.57%	1859	-35.92%
1000	Win32	3296	1968	-40.29%	2531	-23.21%
1000	Win64	4469	1989	-55.49%	2047	-54.20%

Comments about Benchmarks

Find below some comments about the results obtained after benchmark the 3 different memory managers:

- Using in single threaded application, there are no big differences in performance between FastMM4, FastMM5 and FasMM4-AVX.
- **FastMM5** and **FastMM4-AVX** work much **better** in **multithreaded** applications.
- The **differences** between FastMM5 and FastMM4-AVX are **small**, at least doing these benchmarks.
- **Windows 32** benchmarks performs **better** than **Windows 64** tests. Using FastMM5 or FastMM4-AVX in a Windows 64 applications improves performance more than in Windows 32.

The final decision to choose one memory manager or another depends of the project, I think there is no single memory manager that works as the best in all conditions, so before choose one or another, test, test and test again to see which performance better for your needs

SSL / TLS

sgcWebSockets secures connections with TLS. You choose which TLS backend (IOHandler) is used via **TLSOptions.IOHandler** on client components and **SSLOptions.IOHandler** on server components. Four backends are available.

- OpenSSL (`iohOpenSSL`) is cross-platform (Windows, Linux, macOS, iOS, Android), requires deploying the OpenSSL libraries, and is available in all editions.
- SChannel (`iohSChannel`) is Windows only, native, has no libraries to deploy, and is available in the Professional and Enterprise editions.
- Native Android TLS (`iohAndroidTLS`) is Android only, native, has no OpenSSL .so to deploy, and is available in the Enterprise edition.
- Native Apple TLS (`iohAppleTLS`) targets iOS and macOS, is native, has no OpenSSL .dylib to deploy, and is available in the Enterprise edition.

Backend	Platforms	Library to deploy	Edition
OpenSSL	All platforms	OpenSSL libs	All editions
SChannel	Windows	None	Professional and Enterprise
Native Android TLS	Android	None	Enterprise
Native Apple TLS	iOS, macOS	None	Enterprise

How to select a backend

Set the TLS flag and pick the IOHandler on the component:

```
Client.TLS := True;
Client.TLSOptions.IOHandler := iohAppleTLS; // or iohOpenSSL, iohSChannel, iohAndroidTLS
```

The same TLSOptions API (VerifyCertificate, RootCertFile, CertFile, Password, ALPNProtocols) works across all backends.

SChannel

SChannel uses Microsoft's native TLS stack (Secure Channel / SSPI), built into Windows. No OpenSSL libraries need to be deployed, and it uses the Windows certificate store. Windows only. Available in the Professional and Enterprise editions.
Enable it on a client component:

```
Client.TLS := True;
Client.TLSOptions.IOHandler := iohSChannel;
```

Server components use:

```
Server.SSLOptions.IOHandler := iohSChannel;
```

Certificate validation. Like the other IOHandlers, SChannel does not validate the server certificate unless asked to, and `TLSOptions.VerifyCertificate` defaults to `False`. An application connecting to a server whose certificate is self-signed, expired, or issued for a different host will complete the handshake without any warning until this is set:

```
Client.TLSOptions.VerifyCertificate := True;
```

With `VerifyCertificate := True`, SChannel validates the certificate chain against the Windows certificate store and checks that the certificate matches the host being connected to. Use the OnSChannelVerifyPeer event to inspect the chain yourself or accept a specific certificate that would otherwise be rejected, for example a self-signed one used only for testing.

Certificate revocation

`VerifyCertificate` establishes that the chain is trusted and that it names the host being connected to. It says nothing about whether the certificate has been revoked since it was issued, so a stolen key whose certificate was revoked but has not yet expired still authenticates. Revocation is checked only when it is asked for, through `TLSOptions.SChannel_Options.Revocation`:

```
Client.TLSOptions.VerifyCertificate := True;
Client.TLSOptions.SChannel_Options.Revocation.Check := scrcChainExcludeRoot;
```

The check runs while the certificate chain is built, which means the CRL or OCSP responder named by the certificate is contacted during the handshake. This is why it is off by default. The properties are:

`Check`	How much of the chain is checked. `scrcDisabled` (default) checks nothing. `scrcEndCertificate` checks only the server certificate, the cheapest option. `scrcChainExcludeRoot` checks the server certificate and the intermediates, and is the usual choice. `scrcChain` also checks the root, which a self-signed root rarely publishes a CRL for.
`Timeout`	Milliseconds allowed for the whole revocation retrieval, not per URL, so an unreachable responder cannot stall the handshake. Default 5000. Zero leaves the Windows default in force, which can be considerably longer.
`CacheOnly`	Uses only revocation data Windows has already cached and never goes to the network, so the handshake is never delayed. A cold cache then leaves the status unknown, which `IgnoreNoRevocationCheck` decides. Default `False`.

`IgnoreRevocationOffline`	Accepts the certificate when the responder could not be reached. Default `True`, which is the soft-fail behaviour browsers use. Set it to `False` to reject instead, which is safer but makes every connection depend on the responder being reachable.
`IgnoreNoRevocationCheck`	Accepts a certificate that publishes no CRL distribution point and no OCSP responder, so its status cannot be established at all. Default `True`.

Neither Ignore property ever accepts a certificate that came back confirmed revoked. That outcome is always rejected, and the connection fails with `CRYPT_E_REVOKED`, which OnSChannelVerifyPeer reports as `ecCryptRevoked`. An unreachable responder reports `ecCryptRevocationOffline` and a certificate with no revocation information reports `ecCryptNoRevocationCheck`.
Setting `TLSOptions.Preset := tlspSecureDefaults` turns revocation on as `scrcChainExcludeRoot`, together with the other hardened defaults, and leaves a scope you chose yourself untouched.

TLS version floor and ceiling

The accepted TLS versions are set by two properties. `TLSOptions.Version` is the maximum version, and `TLSOptions.SChannel_Options.VersionMin` is the minimum. Both default to `tlsUndefined`.
Leaving both at `tlsUndefined` states no application policy at all, so Windows decides with its own defaults. That is what Microsoft recommends for new development, and it is what the components ship with.
Setting only the floor is the configuration most applications want:

```
Client.TLS := True;
Client.TLSOptions.IOHandler := iohSChannel;
Client.TLSOptions.SChannel_Options.VersionMin := tls1_2;
```

This reads as "TLS 1.2 or better, up to whatever the operating system supports". The same code negotiates TLS 1.3 on Windows 11 and Windows Server 2022, TLS 1.2 on Windows 10, and refuses TLS 1.0 and TLS 1.1 on both, with no OS detection in the application code.
The combinations are:

`Version` only	Pins an exact version, because it becomes both ends of the range. `Version := tls1_2` accepts TLS 1.2 and nothing else.
`VersionMin` only	A floor with the ceiling left open, so anything from that version up to the highest the system supports is accepted.
Both	A closed range, for example `VersionMin := tls1_2` with `Version := tls1_3`.
Neither	No application policy, the Windows defaults apply. This is the shipped default.

A `VersionMin` above `Version` is an inverted range. It is rejected with an error when the connection starts, it is never silently swapped.

How the range is enforced

The range is enforced twice. It is first applied to the credentials: on `SCH_CREDENTIALS`, the structure used on Windows 10 1809 and later, the versions outside the range are placed in `grbitDisabledProtocols`. It is checked again after the handshake, where the negotiated protocol is read back from `SECPKG_ATTR_CONNECTION_INFO` and the connection is dropped when it falls outside the configured range.
The second check is what makes this a policy rather than a request. The machine wide registry settings take precedence over the credential flags, and the legacy credential path cannot disable a protocol at all, so what was asked for in the credentials is not by itself a guarantee of what was negotiated.

TLS 1.3 on Windows

Schannel supports TLS 1.3 only on Windows 11 and Windows Server 2022 or later. No Windows 10 build has it, including 22H2. TLS 1.3 also requires the `SCH_CREDENTIALS` structure, so requesting exactly `tls1_3` while `SChannel_Options.UseLegacyCredentials` is `True`, or on Windows older than 1809, fails with a clear error instead of quietly downgrading to TLS 1.2.
Registry policy can always restrict further what the application asks for, and it can never widen it. A machine where TLS 1.2 has been disabled in the registry will not negotiate it whatever these properties are set to.
Setting `TLSOptions.Preset := tlspSecureDefaults` applies this same floor, `VersionMin := tls1_2` with the ceiling left open, together with the other hardened settings. It is the one-property form of the configuration shown above.

Native Android TLS

On Android you can use the platform native TLS stack (javax.net.ssl.SSLEngine) instead of OpenSSL, so no libssl.so / libcrypto.so need to be deployed. It validates against the Android system trust store, verifies the host-name, negotiates TLS 1.3, and supports ALPN on Android 10 (API 29) and later. Android only. Enterprise edition. Requires Rad Studio XE8 or later.
iohAndroidTLS is a client only handler. A server with **SSLOptions.IOHandler** = iohAndroidTLS raises an exception when it is activated, so use iohNativeTLS or iohOpenSSL on a server. HTTP/2 clients negotiate h2 through ALPN with this handler too.
Enable it on a client component:

```
Client.TLS := True;
Client.TLSOptions.IOHandler := iohAndroidTLS;
```

Native Apple TLS

On iOS and macOS you can use Apple native TLS instead of OpenSSL, so no libssl.dylib / libcrypto.dylib need to be deployed. It auto-selects the best system API: Network.framework (TLS 1.3) on macOS 10.14+ / iOS 12+, falling back to Secure Transport (TLS 1.2) on older systems. It supports the system trust store, SNI / hostname verification, an OnVerifyPeer callback, a custom CA (RootCertFile), client certificate / mutual TLS (CertFile + Password) and ALPN. Enterprise edition. Requires Rad Studio XE6 or later.
iohAppleTLS is a client only handler. A server with **SSLOptions.IOHandler** = iohAppleTLS raises an exception when it is activated, so use iohNativeTLS or iohOpenSSL on a server. HTTP/2 clients negotiate h2 through ALPN with this handler too.
Enable it on a client component:

```
Client.TLS := True;
Client.TLSOptions.IOHandler := iohAppleTLS;
Client.TLSOptions.ALPNProtocols.Add('http/1.1');
Client.TLSOptions.RootCertFile := '';   // optional custom CA
Client.TLSOptions.CertFile := '';        // optional client certificate (PKCS#12) for mTLS
Client.OnAppleTLSVerifyPeer := DoVerifyPeer; // optional
```

Running without OpenSSL

sgcWebSockets can run with no OpenSSL library at all. Two independent settings make that possible, and both are built on the pure Pascal sgcCrypto units, which ship from the Standard edition up.

- **iohNativeTLS** is an IOHandler value. It replaces OpenSSL for the connections of a component: TLS, QUIC and DTLS. See Native TLS.
- **sgcCryptoBackend** is a runtime switch. It selects the backend of the application cryptography, everything that is not a connection: hashes, signatures, key agreement and certificate handling.

An application that needs OpenSSL neither for its connections nor for its other cryptography uses both.

1. Runtime switch: sgcCryptoBackend

The unit **sgcBase_Helpers** declares the global variable **sgcCryptoBackend** of type **TsgcCryptoBackend**:

- **cbOpenSSL** (default): the application cryptography runs on OpenSSL, as in previous versions.
- **cbNative**: it runs on sgcCrypto and OpenSSL is never loaded for it.
- **cbAuto**: OpenSSL is used when it can be loaded, sgcCrypto when it cannot. The check runs once and the result is cached.

The switch is global, not per component. Set it once at startup, before any component is created. **sgcUseNativeCrypto** returns True when the native backend is in use. The native backend needs the sgcCrypto units compiled in (SGC_CRYPTO). Without them, cbNative raises the exception "Native crypto not available, sgcCrypto units are not compiled in". Call **sgcCheckNativeCrypto** to get that error at startup instead of on first use.

```
uses
  sgcBase_Helpers;

begin
  sgcCryptoBackend := cbNative;
  sgcCheckNativeCrypto;
  Application.Initialize;
  ...
```

These parts follow the switch:

Area	What runs natively
Hash and HMAC helpers	The hash and HMAC functions of sgcBase_Helpers, and sgcEncryptString / sgcDecryptString.
JWT, JWS signatures	HMAC, RSA, RSA-PSS, ECDSA and EdDSA signing and verification (sgcSSL_RSA, sgcSSL_PS, sgcSSL_ES, sgcSSL_ED). sgcES_GetCurveFromPublicKey works only on an OpenSSL key and raises on the native backend.
Certificates	The sgcSSL helpers: PEM and DER conversion, certificate loading, CRL URI and certificate chain verification.
OAuth2 and DPoP	The OAuth2 JWK helper, and the DPoP proof check of the OAuth2 server (ES256, ES384, ES512, RS256, RS384, RS512). See DPoP.
Amazon	AWS Signature V4, the CloudFront signed URL, and the Signature V4 authentication of the Amazon IoT MQTT client, whose SignatureV4.OpenSSL_Options are then ignored.
Web Push	The VAPID ES256 token, the ECDH key agreement, HKDF and the AES-128-GCM payload encryption. Only an https connection to the push service loads a TLS library, through its IOHandler.
E2EE	Key agreement, AES-GCM and the random generator. The public keys go on the wire as PEM on both backends, raw keys are still accepted, so peers on different backends interoperate.
WebAuthn, SAML, OIDC	The WebAuthn certificate, COSE key and attestation checks, and the XMLDSig signature check used by SAML and OIDC.
NTLM	The MD4 and DES code of NTLM authentication. It is native in every build and no longer needs any define.

AEAD, KEM, HKDF	sgcSSL_AEAD (AES-128-GCM, AES-256-GCM, ChaCha20-Poly1305), sgcSSL_KEM (ML-KEM-768 and ECDH P-256) and sgcSSL_HKDF (now also with sgcHKDF_MD_SHA384). ML-KEM-768 does not need OpenSSL 3.5, and the keys use the same encodings as OpenSSL 3.5. In a FIPS build ChaCha20-Poly1305 is not available and raises.
QUIC, DTLS, SRTP	QUIC packet protection, the DTLS certificate and SRTP. With iohNativeTLS they are already native per connection, whatever the switch says.

sgcCryptoBackend does not change the TLS connection of a component. A component that keeps IOHandler = iohOpenSSL still loads OpenSSL when it connects.

2. Optional define: SGC_NATIVE_CRYPTO

The define is off by default and is no longer needed to run without OpenSSL. Without it, both the OpenSSL and the native code are compiled and sgcCryptoBackend decides at runtime. With it, the native backend is forced whatever sgcCryptoBackend says, and the OpenSSL code is removed from the application crypto units, so the executable is smaller. To enable it, open **sgcVer.inc** and remove the dot from the line

```
{.$DEFINE SGC_NATIVE_CRYPTO}
```

or add SGC_NATIVE_CRYPTO to the conditional defines of your project. SGC_NATIVE_CRYPTO also turns on SGC_CRYPTO, the sgcCrypto units it needs. DISABLE_SGC_NATIVE_CRYPTO turns it off again.

3. Zero OpenSSL at runtime

Which settings you need depends on what the application uses:

Area	Settings for no OpenSSL
TLS over TCP: WebSocket, HTTP/1, HTTP/2, TCP, MQTT and the other protocols	TLSOptions.IOHandler (client) or SSLOptions.IOHandler (server) = iohNativeTLS. See Native TLS.
QUIC and HTTP/3	IOHandler = iohNativeTLS. The handshake, the packet protection, key update and the Retry tag are all native. See QUIC and HTTP/3.
DTLS and WebRTC	DTLSOptions.IOHandler = iohNativeTLS. The handshake, the DTLS certificate and the SRTP keys are native. See DTLSOptions and WebRTC DTLS.
Application crypto: JWT, OAuth2, Web Push, Amazon Signature V4, E2EE, WebAuthn, SAML and the other areas of the first table	sgcCryptoBackend := cbNative at startup, or the SGC_NATIVE_CRYPTO define.

The iohNativeTLS rows need no define and no sgcCryptoBackend setting. Selecting iohNativeTLS in a build without the sgcCrypto units raises an exception, it never falls back to OpenSSL silently.

iohNativeTLS

With iohNativeTLS the TLS 1.3 and TLS 1.2 handshake and the record layer run on the native engine. It works for the WebSocket, HTTP/1, HTTP/2 and TCP clients and servers, and for everything built on them: MQTT, AMQP, STOMP and the other protocols, the API components, the IoT clients (**Certificate.IOHandler**), the upstream connection of the proxy server (**Proxy.TLSOptions**) and the LDAP client. The certificate may be PEM, DER or a .pfx / .p12 file.

OpenSSL

OpenSSL is a software library for applications that secure communications over computer networks against eavesdropping or need to identify the party at the other end. It is widely used by Internet servers, including the majority of HTTPS websites.

This library is required by components based on Indy Library when a secure connection is needed. If your application requires OpenSSL, you must have necessary files in your file system before deploying your application:

Currently, sgcWebSockets supports: **1.0.2**, **1.1** and **3.0 to 3.3 openSSL** versions.

Platform	API 1.0	API 1.1	API 3.*	Static/Dynamic Linking
Windows (32-bit and 64-bit)	libeay32.dll and ssleay32.dll	libcrypto-1_1.dll and libssl-1_1.dll	libcrypto-3.dll and libssl-3.dll	Dynamic
OSX	libcrypto.dylib, libssl.dylib	libcrypto.1.1.dylib, libssl.1.1.dylib	libcrypto.3.dylib, libssl.3.dylib	Dynamic
iOS Device (32-bit and 64-bit)	libcrypto.a and libssl.a	libcrypto.a and libssl.a	libcrypto.a and libssl.a	Static
iOS Simulator	libcrypto.dylib, libssl.dylib	libcrypto.1.1.dylib, libssl.1.1.dylib	libcrypto.3.dylib, libssl.3.dylib	Dynamic
Android Device	libcrypto.so, libssl.so	libcrypto.so, libssl.so	libcrypto.so, libssl.so	Dynamic

Find below how to **configure OpenSSL** libraries for each platform:

- Windows
- OSX
- Android
- iOS

Static linking: instead of loading the OpenSSL DLLs at runtime, OpenSSL can be linked directly inside your executable, so no DLL needs to be deployed. Read more at OpenSSL Static Linking.

openSSL Configurations

sgcWebSockets Indy-based components allow you to configure some OpenSSL properties. Access to the following properties:

- **Server Components:** SSLOptions.OpenSSL_Options.
- **Client Components:** TLSOptions.OpenSSL_Options.

API Version

Standard Indy library only allows loading **1.0.2 OpenSSL** libraries; these libraries have been deprecated and the latest OpenSSL releases use the 1.1.1 API.
sgcWebSockets Enterprise allows you to load **1.1.1 openSSL** libraries, you can configure in this property which openSSL API version will be loaded. Only one API version can be loaded by process (so you can't mix openSSL 1.0.2 and 1.1.1 libraries in the same application).

LibPath

This property allows you to set the location of openSSL libraries. This is useful for Android or OSX projects, where the location of the openSSL libraries must be set.
Accepts the following values:

- **oslpNone:** this value doesn't set any library path value (is the value by default).
- **oslpDefaultFolder:** this value sets the default folder of openSSL libraries. This path is different for every personality (windows, osx...).

Load Additional OpenSSL Functions

Use a callback to load additional OpenSSL functions not defined by default. You can read more at OpenSSL Load Additional Functions.

Ciphers

If you want to provide support for TLS 1.2 and 1.3 on your server and using the best security and performance, use the following configuration:

SSLOptions.Version := tls1_3;
SSLOptions.OpenSSL_Options.VersionMin := tls1_2;
SSLOptions.OpenSSL_Options.APIVersion := oslAPI_3_0;

And set the following cipher list.

AEAD-AES128-GCM-SHA256:AEAD-AES256-GCM-SHA384:AEAD-CHACHA20-POLY1305-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384

Self-Signed Certificates

You can use self-signed certificates for testing purposes. You only need to execute the following command to create a self-signed certificate:

```
openssl req -newkey rsa:2048 -new -nodes -x509 -days 3650 -keyout key.pem -out cert.pem
```

It will create 2 files: cert.pem (certificate) and key.pem (private key). You can combine both files into a single one. Just create a new file and copy the content of both files into it. So you will have a structure like this:

```
-----BEGIN PRIVATE KEY-----
....
-----END PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
....
-----END CERTIFICATE-----
```

Common Errors

SSL_GET_RECORD: wrong version number

This error means that the server and the client are using different versions of the SSL/TLS protocol. To fix it, try to set the correct version in the server and/or client component.

```
Server.SSLOptions.Version
Client.TLSOptions.Version
```

SSL3_GET_RECORD: decryption failed or bad record mac

Usually this error is raised when:

1. Check that you are using the latest OpenSSL version. If it is too old, update to the latest supported version.
2. If this error appears randomly, it is usually because more than one thread is accessing the OpenSSL connection. You can try to set NotifyEvents = neNoSync which means that the events: OnConnect, OnDisconnect, OnMessage... will be fired in the context of thread connection, this avoids some synchronization problems and provides better performance. As a down side, if for example you are updating a visual control in a form when you receive a message, you must implement your own synchronization methods because visual controls are not thread-safe.

OpenSSL | Windows

There is one version for 32 bits and another for 64 bits. You must copy these libraries in the same folder where is your application or in your system path.
If your Operating System is Windows 32 bits, just copy in System32 folder.
If your Operating System is Windows 64 bits, copy 64 bits version in System32 folder and 32 bits version in SysWow64 folder.

API 1.0

Requires the following libraries:

- libeay32.dll
- ssleay32.dll

If your Operating System is Windows 32 bits, just copy in System32 folder.
If your Operating System is Windows 64 bits, copy 64 bits version in System32 folder and 32 bits version in SysWow64 folder.

You can download latest libraries from your account (libraries don't have external dependencies and are digitally signed).

API 1.1

Requires the following libraries:

Windows 32

- libcrypto-1_1.dll
- libssl-1_1.dll

Windows 64

- libcrypto-1_1-x64.dll
- libssl-1_1-x64.dll

If your Operating System is Windows 32 bits, just copy in System32 folder.
If your Operating System is Windows 64 bits, copy 64 bits version in System32 folder and 32 bits version in SysWow64 folder.

You can download latest libraries from your account (libraries don't have external dependencies and are digitally signed).

API 3.*

Requires the following libraries:

Windows 32

- libcrypto-3.dll
- libssl-3.dll

Windows 64

- libcrypto-3-x64.dll
- libssl-3-x64.dll

If your Operating System is Windows 32 bits, just copy in System32 folder.
If your Operating System is Windows 64 bits, copy 64 bits version in System32 folder and 32 bits version in SysWow64 folder.

You can download latest libraries from your account (libraries don't have external dependencies and are digitally signed).

If you're using a p12 certificate, requires to deploy the legacy.dll library. Read more about OpenSSL p12 Certificates.

OpenSSL | OSX

Newer versions of OSX don't include openssl libraries or are too old, so you must deploy with your application. Deploy these libraries using following steps:

- Open Project/Deployment in your project.
- Add required libraries.
- Set RemotePath = 'Contents\Macos\'.
- Configure the openSSL LibPath to default folder:
    - Client.TLSOptions.OpenSSL_Options.LibPath = oslpDefaultFolder.
    - Server.SSLOptions.OpenSSL_Options.LibPath = oslpDefaultFolder.

API 1.0

Requires the following libraries:

- libcrypto.dylib
- libssl.dylib

You can download latest libraries from your account.

API 1.1

Requires the following libraries:

- libcrypto.1.1.dylib
- libssl.1.1.dylib

There is one version for 32 bits and another for 64 bits. You must copy these libraries in the same folder where your application is located.
You can download latest libraries from your account.

API 3.0

Requires the following libraries:

- libcrypto.3.dylib
- libssl.3.dylib

Only the 64-bit version is provided. You must copy these libraries in the same folder where your application is located.
You can download latest libraries from your account.

If you include the OpenSSL libraries in an OSX application, after the application has been Notarized, the libraries will be signed, you can check this using the following command:

```
codesign -dv --verbose=4 libcrypto.1.1.dylib
```

Check the following video which shows how to build a MacOSX64 Application with OpenSSL libraries.

https://www.esegece.com/websockets/videos/delphi/quickstart/275-build-macosx64-application/file

Errors

Clients should not load the unversioned libcrypto dylib as it does not have a stable ABI.

On MacOS Monterey+, you can get this error trying to load the openSSL libraries, the error happens when tries to load first the openSSL libraries without version (libcrypto.dylib for example).
To fix this error set in the property **OpenSSL_Options.UnixSymLinks** the value **osIsSymLinksDontLoad**. This avoids the loading of the openSSL libraries without version.

OpenSSL | Android

Newer versions of Android don't include openssl libraries or are too old, so you must deploy with your application. Deploy these libraries using following steps:

- Open Project/Deployment in your project.
- Add required libraries.
- Set RemotePath = '.\assets\internal'.
- Configure the openSSL LibPath to default folder:
    - Client.TLSOptions.OpenSSL_Options.LibPath = oslpDefaultFolder.
    - Server.SSLOptions.OpenSSL_Options.LibPath = oslpDefaultFolder.

API 1.0

Requires the following libraries:

- libcrypto.so
- libssl.so

You can download latest libraries from your account.

On **Android 64bits**, using TLS 1.2 may raise the following error:

INT_RSA_VERIFY:bad signature

This is an OpenSSL error that is fixed in API 1.1.
You can try to use TLS 1.0 or TLS 1.1 (if the server still supports these encryption methods to avoid this error).

API 1.1

Requires the following libraries:

- libcrypto.so
- libssl.so

You can download latest libraries from your account.

API 3.0

Requires the following libraries:

- libcrypto.so
- libssl.so

You can download latest libraries from your account.

OpenSSL | iOS

To install OpenSSL in a 64-bit iOS device, you must copy the libcrypto.a and libssl.a SSL library files to your system. Download the .zip iOS OpenSSL, extract it and find the .a files in the \lib directory. You must copy the libcrypto.a and libssl.a SSL library files to these directories:

- C:\Program Files (x86)\Embarcadero\Studio\<IDE Version>\lib\iosDevice64\debug
- C:\Program Files (x86)\Embarcadero\Studio\<IDE Version>\lib\iosDevice64\release

Add **sgcIdSSLOpenSSLHeaders_static** (or IdSSLOpenSSLHeaders_static if your sgcWebSockets edition is not Enterprise) unit to your uses clause.

If you need to **deploy** any file, you can set RemotePath = StartUp\Documents and to load the file use (requires add System.IOUtils to uses clause):

TPath.GetDocumentsPath + PathDelim + <your filename>

The openSSL libraries must not be deployed using the menu Project/Deployment under iOS.

API 1.1

Modify **IdCompilerDefines.inc** and enable SGC_OPENSSL_API_1_1 in IOS section:

```
{$IFDEF IOS}
  {$DEFINE HAS_getifaddrs}
  {$DEFINE USE_OPENSSL}
  {$IFDEF CPUARM}
    // RLebeau: For iOS devices, OpenSSL cannot be used as an external library,
    // it must be statically linked into the app.  For the iOS simulator, this
    // is not true.  Users who want to use OpenSSL in iOS device apps will need
    // to add the static OpenSSL library to the project and then include the
    // IdSSLOpenSSLHeaders_static unit in their uses clause. It hooks up the
    // statically linked functions for the IdSSLOpenSSLHeaders unit to use...
    {$DEFINE STATICLOAD_OPENSSL}
// sgc--> enable for openssl API 1.1
{$DEFINE SGC_OPENSSL_API_1_1}
  {$ENDIF}
{$ENDIF}
```

You can download libraries from your account.

API 3.0

Modify **IdCompilerDefines.inc** and enable SGC_OPENSSL_API_1_1 and SGC_OPENSSL_API_3_0 in IOS section:

```
{$IFDEF IOS}
  {$DEFINE HAS_getifaddrs}
  {$DEFINE USE_OPENSSL}
  {$IFDEF CPUARM}
    // RLebeau: For iOS devices, OpenSSL cannot be used as an external library,
    // it must be statically linked into the app.  For the iOS simulator, this
    // is not true.  Users who want to use OpenSSL in iOS device apps will need
    // to add the static OpenSSL library to the project and then include the
    // IdSSLOpenSSLHeaders_static unit in their uses clause. It hooks up the
    // statically linked functions for the IdSSLOpenSSLHeaders unit to use...
    {$DEFINE STATICLOAD_OPENSSL}
```

```
    // sgc--> enable for openssl API 1.1
    {$DEFINE SGC_OPENSSL_API_1_1}
    // sgc--> enable for openssl API 3.0
    {$DEFINE SGC_OPENSSL_API_3_0}
    {$ENDIF}
  {$ENDIF}
```

You can download libraries from your account.

OpenSSL | Own CA Certificates

Github post

To create a certificate signed by your own CA and that can be trusted by Web Browsers (like Chrome) after adding CA certificate to local machine.

1. Prepare the configuration files for creating certificates without prompts

CA.cnf

```
[ req ]
prompt = no
distinguished_name = req_distinguished_name
[ req_distinguished_name ]
C = US
ST = Localzone
L = localhost
O = Certificate Authority Local Center
OU = Develop
CN = develop.localhost.localdomain
emailAddress = root@localhost.localdomain
```

localhost.cnf

```
[req]
default_bits  = 2048
distinguished_name = req_distinguished_name
req_extensions = req_ext
x509_extensions = v3_req
prompt = no
[req_distinguished_name]
countryName = US
stateOrProvinceName = Localzone
localityName = Localhost
organizationName = Certificate signed by my CA
commonName = localhost.localdomain
[req_ext]
subjectAltName = @alt_names
[v3_req]
subjectAltName = @alt_names
[alt_names]
IP.1 = 127.0.0.1
IP.2 = 127.0.0.2
IP.3 = 127.0.0.3
IP.4 = 192.168.0.1
IP.5 = 192.168.0.2
IP.6 = 192.168.0.3
DNS.1 = localhost
DNS.2 = localhost.localdomain
DNS.3 = dev.local
```

2. Generate a CA private key and Certificate (valid for 5 years)

```
openssl req -nodes -new -x509 -keyout CA_key.pem -out CA_cert.pem -days 1825 -config CA.cnf
```

3. Generate web server secret key and CSR

```
openssl req -sha256 -nodes -newkey rsa:2048 -keyout localhost_key.pem -out localhost.csr -config localhost.cnf
```

4. Create certificate and sign it by own certificate authority (valid 1 year)

```
openssl x509 -req -days 398 -in localhost.csr -CA CA_cert.pem -CAkey CA_key.pem -CAcreateserial -out localhost_ce
```

5. Output files will be:

- `CA.cnf` → OpenSSL CA config file. May be deleted after certificate creation process.
- `CA_cert.pem` → [Certificate Authority] certificate. This certificate must be added to the browser local authority storage to make trust all certificates that created with using this CA.
- `CA_cert.srl` → Random serial number. May be deleted after certificate creation process.
- `CA_key.pem` → Must be used when creating new [localhost] certificate. May be deleted after certificate creation process (if you do not plan reuse it and CA_cert.pem).
- `localhost.cnf` → OpenSSL SSL certificate config file. May be deleted after certificate creation process.
- `localhost.csr` → Certificate Signing Request. May be deleted after certificate creation process.
- `localhost_cert.pem` → SSL certificate. **Must be configured in SSLOptions.CertFile property of the server.**
- `localhost_key.pem` → Secret key. **Must be installed at SSLOptions.KeyFile property of the server.**

OpenSSL | P12 Certificates

OpenSSL 3.0 moved several deprecated or insecure algorithms into an internal library module called legacy provider. It is not loaded by default, so apps (or their language runtimes) that use OpenSSL for cryptographic operations cannot use such algorithms when loading certificates, creating message digests ...

Algorithms in the legacy provider include MD2, MD4, MDC2, RMD160, CAST5, BF (Blowfish), IDEA, SEED, RC2, RC4, RC5 and DES (but not 3DES).

For security reasons, it is strongly recommended to retire the use of these legacy algorithms.

If your application utilizes client certificates stored in a file encrypted with a legacy cipher such as RC2-40-CBC, it is possible to "modernize" the certificate file by re-encrypting it using the openssl program.

For example, if you have a client.p12 (or client.pfx) certificate file on your local computer:

```
$ openssl pkcs12 -legacy -in client.p12 -nodes -out cert-decrypted.tmp
(enter passphrases if prompted)

$ openssl pkcs12 -in cert-decrypted.tmp -export -out client-new.p12
(enter passphrases if prompted)

$ rm cert-decrypted.tmp
```

The exported client-new.p12 certificate file now contains the same keys, but encrypted using AES-256-CBC.

Check below the configuration for sgcWebSockets and sgcIndy packages:

sgcWebSockets

- Set the property **OpenSSL_Options.Legacy.Enabled** to True.
- Set the location of the Legacy library.
    - **OpenSSL_Options.Legacy.LibPath:** here you can configure where is located the legacy library
        - **oslpNone:** this is the default, the legacy library should be in the same folder where is the binary or in a known path.
        - **oslpDefaultFolder:** sets automatically the legacy library path where the libraries should be located for all IDE personalities.
        - **oslpCustomFolder:** if this is the option selected, define the full path in the property LibPathCustom.
    - **OpenSSL_Options.Legacy.LibPathCustom:** when LibPath = oslpCustomFolder define here the full path where are located the legacy library.

sgcIndy

- Set the property **SSLOptions.Legacy** to True.
- Before start the server or client, set the path where the legacy.dll library it's located. Use the function **IdOpenSSLSetOSSLPath** and pass the path as argument.

OpenSSL | Verify Certificate

When using OpenSSL and setting the option Verify Certificate, the following error may appear:

```
Error connecting with SSL.error:80000002:system library::No such file or directory.
```

If you handle the event OnVerifyPeer and the parameter Error has a value of 20, the error means:

```
X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY
```

The main reason for this error is one or more certificates presented by the remote server are not present in the certificate store of your application. To resolve this, you can use the property RootCertFile and set the path where the CA file is located. If you don't have any, you can download from mozilla for example:

https://curl.haxx.se/docs/caextract.html

After setting the RootCertFile, the previous error should be gone.

What VerifyCertificate checks

Setting **VerifyCertificate** to True validates the certificate chain against the trusted roots *and* binds the certificate to the host you are connecting to, so a chain-valid certificate issued for a different host is rejected with X509 error 62, `X509_V_ERR_HOSTNAME_MISMATCH`. This matches the SChannel and Apple backends, which have always checked the host name under this same property.

Two details are worth knowing:

- The host name is bound only when you connect by **name**. Connecting to a bare IP address binds nothing, because the check is a DNS name comparison. If you connect by IP and want the address itself verified, set **OpenSSL_Options.X509Checks.Mode** to include **oslx509chIPAddress** and make sure the server certificate carries a matching IP SAN.
- **VerifyDepth** limits how far the chain walk may go. Left at its default it allows an ordinary CA-signed chain, so set it explicitly only when you need to constrain the depth; a value that is too low rejects the issuer and the handshake fails with the error 20 described above.

OpenSSL | Load Additional Functions

By default, Indy defines the most common OpenSSL functions needed to encrypt communications, but sometimes you need more functions for encryption, signing, etc. You can use the method IdOpenSSLSetLoadFuncsCallback to assign a callback for loading additional OpenSSL functions dynamically.

IdOpenSSLSetLoadFuncsCallback

```
TIdLoadSSLFuncsCallback = procedure(TIdLibHandle hIdSSL, TIdLibHandle hIdCrypto, const TStringList* FailedLoadLis
```

This is a procedure type that serves as a callback, it takes three parameters:

- **hIdSSL:** TIdLibHandle - Handle to the loaded SSL library.
- **hIdCrypto:** TIdLibHandle - Handle to the loaded Crypto library.
- **FailedLoadList:** TStringList - A list of functions that failed to load.

The purpose of this callback is to allow the user to perform custom processing when OpenSSL functions are being loaded, such as logging failed function loads or handling errors.

IdOpenSSLSetUnLoadFuncsCallback

```
TIdLoadSSLFuncsCallback = procedure(TIdLibHandle hIdSSL, TIdLibHandle hIdCrypto, const TStringList* FailedLoadLis
```

It serves as a callback for unloading SSL functions. This is useful for performing cleanup when OpenSSL libraries are being unloaded.

How to load custom function

Find below a simple example of how to load the function EVP_PKEY_CTX_set_rsa_padding using the callbacks.

```
typedef int (__cdecl *TEVP_PKEY_CTX_set_rsa_padding)(void* ctx, int pad);
TEVP_PKEY_CTX_set_rsa_padding EVP_PKEY_CTX_set_rsa_padding = nullptr;

// Mock function for LoadLibFunction (you need to implement this according to your needs)
void* LoadLibFunction(TIdLibHandle hLib, const char* funcName) {
    return GetProcAddress((HMODULE)hLib, funcName);
}

void __fastcall DoOpenSSLLoadFuncsCallback(TIdLibHandle hIdSSL, TIdLibHandle hIdCrypto, const TStringList* Failed
    EVP_PKEY_CTX_set_rsa_padding = (TEVP_PKEY_CTX_set_rsa_padding)LoadLibFunction(hIdCrypto, "EVP_PKEY_CTX_set_rs
}
void __fastcall DoOpenSSLUnLoadFuncsCallback() {
    EVP_PKEY_CTX_set_rsa_padding = nullptr;
}

// Mock function to simulate IdOpenSSLSetLoadFuncsCallback
void IdOpenSSLSetLoadFuncsCallback(void (__fastcall *callback)(TIdLibHandle, TIdLibHandle, const TStringList*)) {
    callback(nullptr, nullptr, nullptr);  // Example call
}
// Mock function to simulate IdOpenSSLSetUnLoadFuncsCallback
void IdOpenSSLSetUnLoadFuncsCallback(void (__fastcall *callback)()) {
    callback();  // Example call
}
```

```
int main() {
    IdOpenSSLSetLoadFuncsCallback(DoOpenSSLLoadFuncsCallback);
    IdOpenSSLSetUnLoadFuncsCallback(DoOpenSSLUnLoadFuncsCallback);
    return 0;
}
```

OpenSSL | Static Linking

OpenSSL can be linked directly inside your application executable, instead of being loaded at runtime from the libcrypto and libssl DLL files. This means an application that uses TLS does not need to deploy any OpenSSL DLL, everything it needs is already inside the .exe.
This is useful when you want to distribute a single file, when you cannot guarantee that the target machine has the right OpenSSL DLL installed, or when your deployment process does not allow extra files next to the executable.

Requirements

- Windows only, both **32 bit** and **64 bit**. It is not available on OSX, Android or iOS.
- **Delphi XE2** or newer. Older Delphi versions cannot link an object file with this format, so on those versions this feature is simply not available and your application keeps loading the OpenSSL DLLs as usual.
- Only available in **sgcWebSockets Enterprise** and **All-Access**, and only when the **custom Indy library** is in use, which is the default for these editions. If you disabled the custom Indy library, for example by removing the SGC_CUSTOM_INDY compiler define, static linking has no effect. Read more at Indy.

How to Configure

There is nothing to set up. Just add the unit `sgcIdSSLOpenSSL_Static` to the uses clause of your project, for example in the main .dpr file:

```
uses
  sgcIdSSLOpenSSL_Static;
```

That is all. There is no compiler directive to enable and no OpenSSL API version to choose, your application starts using the OpenSSL library linked into the executable the next time it runs, for both client and server components. If you remove the unit from the uses clause, your application goes back to loading the OpenSSL DLLs exactly as before.
The OpenSSL version currently linked is **3.5.7**. This version is updated from time to time to include the latest OpenSSL fixes, so you don't need to track or deploy OpenSSL versions yourself.

Common Errors

E2045 Bad object file format
This means you are compiling with a Delphi version older than XE2. Static linking requires Delphi XE2 or newer, remove the sgcIdSSLOpenSSL_Static unit from the uses clause and your application will keep working with the OpenSSL DLLs.

Indy

Indy library is an open source client/server communications library that supports TCP/UDP/RAW sockets, as well as over 100 higher level protocols including SMTP, POP3, IMAP, NNTP, HTTP, FTP, and many more. Indy is written in Delphi but is also available for C++Builder and FreePascal. sgcWebSockets uses Indy as a base for some components and the different sgcWebSockets Editions make a different use of the Indy library.

sgcWebSockets supports protocols like HTTP/2 which require the use of ALPN, can use TLS 1.3 using openSSL 1.1.1 or openSSL 3.0.0... all these features are not supported by the standard Indy library, so sgcWebSockets Enterprise edition includes a custom Indy library that supports these features. To avoid uninstalling the standard Indy library from the IDE, the required Indy files are renamed by adding the prefix "sgc", so for example: the unit "IdGlobal" is renamed to "sgcIdGlobal". This way, both versions can coexist without problems.

Find below which Indy version is used by every sgcWebSockets Edition:

sgcWebSockets Edition	Indy Version
STANDARD	Standard
PROFESSIONAL	Standard
ENTERPRISE	Custom

Customers with "Registered" licenses, which are old licenses from before the sgcWebSockets package was split, will find the following sgcWebSockets package versions:

sgcWebSockets Edition	Indy Version
sgcWebSockets	Standard
sgcWebSockets min	Standard
sgcWebSockets min Indy*	Custom

*The sgcWebSockets_min_indy is the same as sgcWebSockets Enterprise edition.

The use of the custom Indy version is defined in the file "sgcVer.inc" located in the source folder. There is a compiler define called "SGC_CUSTOM_INDY" which enables or disables the use of this indy version. If you have a Enterprise Edition and want to disable the use of the custom indy, just delete the following compiler define:

```
{$DEFINE SGC_CUSTOM_INDY}
```

Of course, if you enable SGC_CUSTOM_INDY but you don't have in the source folder the required custom indy version units, this compiler define won't work.

sgcIndy package

The use of the custom Indy version is not limited to the sgcWebSockets components. Some customers want to make use of the new features of this custom Indy version in standard Indy components like SMTP for example, so they can use TLS 1.3 when sending emails, using FTP servers, etc. The eSeGeCe ALL-ACCESS edition provides an additional full Indy package with all these features. This package, called "sgcIndy package", includes the full Indy library with support for openSSL 1.1.1 and openSSL 3.0.0. So you first must uninstall your current Indy library installed in your IDE and then install this version, the process to install the sgcIndy package is exactly the same as for any Indy library (here the units are not renamed).

When you want to use openSSL libraries, just set the global variable OPENSSL_API_VERSION to the desired OpenSSL API version before loading the OpenSSL libraries. This global variable is in the unit IdSSLOpenSSLHeaders.

Example: to use the openSSL 1.1.1 libraries

```
OPENSSL_API_VERSION := opSSL_1_1;
```

How to use a Single sgcIndy package

When using eSeGeCe ALL-ACCESS and the sgcIndy package in the same application, the sources may be duplicated because the sgcWebSockets Enterprise version uses a custom Indy version with the Indy units renamed. This means that, for example, units like sgcIdGlobal.pas and IdGlobal.pas will be compiled in the same application (the first is used when using any component of the sgcWebSockets Enterprise package and the second when using any component of the sgcIndy package, like ftp, smtp...).

To avoid this behavior, the sgcWebSockets package can be configured to use the installed sgcIndy version and still make use of all the components. To do this, follow the instructions below:

1. Open the file sgcVer.inc, located in the Source folder of the sgcWebSockets package.
2. Disable the compiler directive: SGC_CUSTOM_INDY. This option tells the compiler that the files starting with sgcId*.pas exist and must be used when compiling the sgcWebSockets Enterprise Package.
3. Enable the following compiler: SGC_INDY_LIB. This option tells the compiler that the sgcIndy package is installed and must be used when compiling the package.

```
{$DEFINE SGC_INDY_LIB}
```

Using the previous configuration, the sgcWebSockets Enterprise package will use the sgcIndy package that is installed and all the features that make use of this package (like http/2, IOCP, openSSL 3.0...) will be enabled.

WebSocket Events

WebSocket connections have the following events:

OnConnect
The event raised when a new connection is established.

OnDisconnect
The event raised when a connection is closed.

OnError
The event raised when a connection has any error.

OnMessage
The event raised when a new text message is received.

OnBinary
The event raised when a new binary message is received.

By default, sgcWebSockets uses an **asynchronous** mechanism to raise these events, when any of these events is raised internally, it queues this message and is dispatched by the operating system when is allowed. This behaviour can be modified using a property called **NotifyEvents**, by default **neAsynchronous** is selected, if **neNoSync** is checked then events will be raised without synchronizing with the main thread (if you need to update any VCL control or access to shared resources, then you will need to implement your own synchronizing method).

neNoSync is recommended when:

1. You need to handle a lot of messages in a very short period of time.
2. Your project is built for command line (if you don't set neNoSync, you won't get any event).
3. Your project is a library.

If not, then you can use the default property value of **neAsynchronous**.

Do not free a component, for example a client or a server, inside one of its own event handlers. The component keeps using its internal state after the handler returns, so freeing it there can raise an access violation with any **NotifyEvents** value. Free it later, outside its events.

WebSocket Parameters Connection

Supported by

TsgcWebSocketClient
Java script

Sometimes it is useful to pass parameters from client to server when a new WebSocket connection is established. If you need to pass some parameters to the server, you can use the following property:

Options / Parameters

By default, is set to '/', if you need to pass a parameter like id=1, you can set this property to '/?id=1'

On Server Side, you can handle client parameters using the following parameter:

```
void WSServerConnect(TsgcWSConnection *Connection);
{
  if (Connection->URL == "/?id=1")
  {
    HandleThisParameter;
  }
}
```

Using Javascript, you can pass parameters using connection url, example:

```
<script src="http://localhost/sgcWebSockets.js" type="text/javascript"></script>
<script type="text/javascript">var socket = new sgcWebSocket('ws://localhost/?id=1');</script>
```

Using inside a DLL

If you need to work with Dynamic Link Libraries (DLL) and sgcWebSockets (or console applications), **NotifyEvents** property needs to be set to **neNoSync**.

Windows Service

When running the sgcWebSockets library in a windows service the **NotifyEvents** property must be set to **neNoSync**.

```
void __fastcall TService1::ServiceStart(TService *Sender, bool &Started)
{
    FServer = new TsgcWebSocketHTTPServer(nullptr);
    FServer->Port = 80;
    FServer->Active = true;
}
//---------------------------------------------------------------------------
// OnServiceExecute
void __fastcall TService1::ServiceExecute(TService *Sender)
{
    while (!Terminated)
    {
        ServiceThread->ProcessRequests(true);
    }
}
//---------------------------------------------------------------------------
// OnServiceStop
void __fastcall TService1::ServiceStop(TService *Sender, bool &Stopped)
{
    if (FServer != nullptr)
    {
        delete FServer;
        FServer = nullptr;
    }
}
```

WebBrowser Test

TsgcWebSocketServer implements a built-in Web page where you can test WebSocket Server connection with your favourite Web Browser.

To access this test page, you need to type the following URL:

```
http://host:port/sgcWebSockets.html
```

Example: if you have configured your WebSocket Server on IP 127.0.0.1 and uses port 80, then you need to type:

```
http://127.0.0.1:80/sgcWebSockets.html
```

In this page, you can test the following WebSocket methods:

Open
Close
Status
Send

To disable WebBrowser HTML Test pages, just set in TsgcWebSocketServer.Options.HTMLFiles = false;

Custom Sub-Protocols

A client can request that the server use a specific subprotocol by including the subprotocol name in its handshake. If it is specified, the server needs to include one of the selected subprotocol values in its response for the connection to be established.

In order to create your own subprotocol, you must inherit from TsgcWSProtocol_Client_Base and TsgcWSProtocol_Server_Base in order to create your custom subprotocols.

Authentication

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
Java script (*only URL Authentication is supported)

The WebSocket specification does not define any authentication method, and web browser implementations do not allow sending custom headers on new WebSocket connections.

To enable this feature you need to access the following property:

Authentication/ Enabled

sgcWebSockets implements 3 different types of WebSocket authentication:

Session: the client needs to do an HTTP GET passing a username and password, and if authenticated, the server responds with a Session ID. With this Session ID, the client opens a WebSocket connection passing it as a parameter. You can use a normal HTTP request to get a session id using and passing user and password as parameters

```
http://host:port/sgc/req/auth/session/:user/:password
```

example: (user=admin, password=1234) --> http://localhost/sgc/req/auth/session/admin/1234

This returns a token that is used to connect to server using WebSocket connections:

ws://localhost/sgc/auth/session/:token

URL: the client opens a WebSocket connection passing the username and password as parameters.

```
ws://host:port/sgc/auth/url/username/password
```

example: (user=admin, password=1234) --> http://localhost/sgc/auth/url/admin/1234

Basic: implements Basic Access Authentication, only applies to VCL Websockets (Server and Client) and HTTP Requests (client Web Browsers don't implement this type of authentication). When a client tries to connect, it sends a header using AUTH BASIC specification.

You can define a list of Authenticated users, using **Authentication/ AuthUsers** property. You need to define every item following this schema: user=password. Example:

admin=admin
user=1234
....

There is an event called **OnAuthentication** where you can handle authentication if the user is not in AuthUsers list, client doesn't send an authorization request... You can check User and Password params and if correct, then set Authenticated variable to True. example:

```
void WSServerAuthentication(TsgcWSConnection *Connection, string aUser, string aPassword, bool &Authenticated)
{
  if ((aUser == "John") && (aPassword == "1234"))
```

```
  {
    Authenticated = true;
  }
}
```

Secure Connections

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
Web Browsers

SSL support is based on Indy implementation, so you need to deploy openssl libraries in order to use this feature. TsgcWebSocketClient supports Microsoft SChannel, so there is no need to deploy openssl libraries for windows 32 and 64 bits if SChannel option is selected in WebSocket Client.

Server Side

To enable this feature, you need to enable the following property:

SSL/ Enable

There are other properties that you need to define:

SSLOptions/ CertFile/ KeyFile/ RootCertFile: you need a certificate in .PEM format in order to encrypt websocket communications.

SSLOptions/ Password: this is optional and only needed if the certificate has a password.

SSLOptions/ Port: port used on SSL connections.

Client Side

To enable this feature, you need to enable the following property:

TLS/ Enable

OpenSSL

By default, client and server components based on Indy make use of openSSL libraries when connect to secure websocket servers.
Indy only supports the 1.0.2 OpenSSL API, so API 1.1 is not supported. If you compile sgcWebSockets with our custom Indy library you can make use of API 1.1 and select TLS 1.3 version. Just select in OpenSSL_Options properties which OpenSSL API you would like to use:

- **oslAPI_1_0:** this is the default Indy API, you can use standard Indy package with openssl 1.0.2 libraries.
- **oslAPI_1_1:** only select if you are compiling sgcWebSockets with our custom Indy library (Enterprise Edition). Will use openssl 1.1.1 libraries.
- **oslAPI_3_0:** only select if you are compiling sgcWebSockets with our custom Indy library (Enterprise Edition). Will use openssl 3.0.0 libraries.
    - **ECDHE:** allows you to enable ECDHE for TLS 1.2 (more secure connections).

Events

There are 2 events which can be used to customize your SSL settings:

OnSSLGetHandler

This event is raised before SSL handler is created, you can create here your own SSL Handler (needs to be inherited from TIdServerIOHandlerSSLBase or TIdIOHandlerSSLBase) and set the properties needed

```
properties needed
void OnServerSSLGetHandler(TObject *Sender, TwsSSLHandler aType, ref TIdServerIOHandlerSSLBase
*aSSLHandler)
{
  TCustomSSLHandler aSSLHandler = new TCustomSSLHandler();
  ...
}
```

OnSSLAfterCreateHandler

If no custom SSL object has been created, one is created by default using the OpenSSL handler. You can access the SSL Handler properties and modify them if needed.

```
void OnSSLAfterCreateHandler(TObject *Sender, TwsSSLHandler aType, TIdServerIOHandlerSSLBase
*aSSLHandler)
{
  dynamic_cast <tidserveriohandlesslopenssl>*(aSSLHandler)->SSLOptions->Method = sslvTLSv1_2;
}
```

Microsoft SChannel

From sgcWebSockets 4.2.6 you can use SChannel instead of openssl (only for windows from Windows 7+). This means there is no need to deploy openssl libraries. TLS 1.0 is supported from windows 7 but if you need more modern implementations like TLS 1.2 in Windows 7 you must enable TLS 1.1 and TLS 1.2 in Windows Registry. Requires Delphi 2010 Professional Edition (or Enterprise Edition for Delphi 7, 2007 and 2009).

HeartBeat

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI
TsgcWebSocketClient

On Server components, automatically sends a ping to all active WebSocket connections every x seconds.

On Client components, automatically sends a ping to the server every x seconds.

HeartBeat has the following properties:

- **Enabled:** if true, sends a ping
- **Interval:** is the value in seconds when a ping will be sent. Example: if value is 10, a ping will be sent every 10 seconds
- **Timeout:** is the time it will wait for a response from the server. Example: if the value is 30, it will wait 30 seconds to receive a response before closing the connection.
- **HeartBeatType:** allows customizing how the HeartBeat works
  - **hbtAlways:** sends a ping every x seconds defined in the Interval.
  - **hbtOnlyIfNoMsgRcvInterval:** sends a ping every x seconds only if no messages have been received during the latest x seconds defined in the Interval property.

Customize HeartBeat

Client and server components allow customization of HeartBeat to send custom pings and check that the connection is still alive. The event **OnBeforeHeartBeat** is built exactly for that; it allows you to send a custom message and/or not send the standard ping.

Example: send a message text as a ping every 30 seconds.

```
void OnBeforeHeartBeat(TObject *Sender; const TsgcWSConnection *Connection; ref bool Handled)
{
  Connection->WriteData("ping");
  Handled = true;
}
```

WatchDog

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI
TsgcWebSocketClient

Server

On Server components, automatically restart server after unexpected shutdown. To check if server is active every 60 seconds, just set the following properties:

```
WatchDog.Enabled = true;
WatchDog.Interval = 60;
WatchDog.Attempts = 0;
```

WatchDog.Monitor allows you to verify if new clients can connect to the server. This is done by an internal client that tries to open a WebSocket connection to the server; if it fails, it restarts the server. To monitor whether clients can connect to the server with a time-out of 10 seconds, set the following properties:

```
WatchDog.Enabled = true;
WatchDog.Interval = 60;
WatchDog.Attempts = 0;
WatchDog.Monitor.Enabled = true;
WatchDog.Monitor.TimeOut = 10;
```

Client

On Client components, automatically reconnect to server after unexpected disconnection. To reconnect after a disconnection every 10 seconds, just set the following properties:

```
WatchDog.Enabled = true;
WatchDog.Interval = 10;
WatchDog.Attempts = 0;
```

By default the delay between reconnection attempts is fixed. The client WatchDog can also use exponential backoff with jitter: WatchDog.Backoff selects the strategy (wdbFixed by default, or wdbExponential to multiply the delay by WatchDog.BackoffMultiplier after each failed attempt), WatchDog.MaxInterval caps the delay in seconds (0 = no cap) and WatchDog.Jitter adds a random factor between 0 and 1 to every delay, so many clients do not reconnect at exactly the same time after a server restart. To reconnect starting at 10 seconds, doubling the delay up to 5 minutes with 20% jitter, set the following properties:

```
WatchDog.Enabled = true;
WatchDog.Interval = 10;
WatchDog.Attempts = 0;
WatchDog.Backoff = wdbExponential;
WatchDog.MaxInterval = 300;
WatchDog.Jitter = 0.2;
```

Logs

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient

This is a useful feature that allows debugging WebSocket connections, to enable this, you need to access the following property:

LogFile/ Enabled

Once enabled, every time a new connection is established it will be logged in a text file. On the server component, if the file does not exist it will be created, but you cannot access it until the server is closed. If you want to open the log file while the server is active, the log file needs to be created before starting the server.

Example:

```
127.0.0.1:49854 Stat Connected.

127.0.0.1:49854 Recv 09/11/2013 11:17:03: GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: 127.0.0.1:5414
Origin: http://127.0.0.1:5414
Pragma: no-cache
Cache-Control: no-cache
Sec-WebSocket-Key: 1n598ldHs9SdRfxUK8u4Vw==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: x-webkit-deflate-frame

127.0.0.1:49854 Sent 09/11/2013 11:17:03: HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: gDuzFRzwHBc18P1CfinlvKv1BJc=

127.0.0.1:49854 Stat Disconnected.
0.0.0.0:0 Stat Disconnected.
```

WebSocket Messages

WebSocket frames can be masked, which means that the message logged cannot be read.
When the property **LogFile.UnMaskFrames** = True (by default it's true)

- Messages **sent** by **WebSocket Client** are saved as **unmasked**.
- Messages **received** by **WebSocket Server** are saved **masked** and **unmasked** (the reason is that when the socket reads the buffer, it does not yet know the protocol of the message, so it saves both).

HTTP

Supported by

TsgcWebSocketHTTPServer

TsgcWebSocketHTTPServer is a component that allows you to handle WebSocket and HTTP connections using the same port. It is very useful when you need to set up a server where only the HTTP port is enabled (usually port 80). This component supports all TsgcWebSocketServer features and allows you to serve HTML pages.

You can **serve HTML pages statically**, using **DocumentRoot** property, example: if you save test.html in directory "C:\inetpub\wwwroot", and you set **DocumentRoot** to "C:\inetpub\wwwroot". If a client tries to access to test.html, it will be served automatically, example:

http://localhost/test.html

Or you can **serve HTML or other resources dynamically** by code, to do this, there is an event called **OnCommandGet** that is fired every time a client requests a new HTML page, image, javascript file... Basically, you need to check which document is requesting client (using ARequestInfo.Document) and send a response to client (using AResponseInfo.ContentText where you send response content, AResponse.ContentType which is the type of response and a AResponseInfo.ResponseNo with a number of response code, usually is 200), example:

```
void WSServerCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Document == "/myfile.js")
  {
    AResponseInfo->ContentText = "<script type='text/javascript'>alert('Hello!');</script>";
    AResponseInfo->ContentType = "text/javascript";
    AResponseInfo->ResponseNo = 200;
  }
}
```

To reduce bandwidth for large HTML pages or JSON responses, see HTTP Compression.

Broadcast and Channels

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI

Broadcast method by default sends a message to **all clients connected**, but you can use **channels** argument to filter and **only broadcast message to clients subscribed** to a channel.

Example: your server has 2 types of connected clients, desktop and mobile devices, so you can create 2 channels "desktop" and "mobile".

If you can identify in the OnConnect event of the server whether a client is mobile, you can do something like the following:

```
void OnServerConnect(TsgcWSConnection *Connection)
{
  if (desktop == true)
  {
    dynamic_cast<TsgcWSConnectionServer*>(Connection)->DoSubscribe("desktop");
  }
}
```

First cast Connection to TsgcWSConnectionServer to access subscription methods and if it fits your filter, it will be subscribed to the desktop channel. Subscription to a channel can be done in any event. For example, you can ask the client to tell you if it is mobile or not and send a message from client to server with info about client. Then you can only broadcast to desktop connections:

```
Server->Broadcast("Your text message", "desktop");
```

If you have 100 connections and 30 are mobile, the message will only be sent to the other 70.

Multiple channels and wildcards

A connection can be subscribed to several channels in one call by separating them with a comma, both from TsgcWSPClient_sgc and from server code:

Channel names are matched as an **exact string** by default, so MQTT-style wildcards such as "topic1/#" or "topic1/+" are not interpreted, they are just literal characters. A client subscribed to "topic1/#" will not receive a message broadcast to "topic1/sensorA".

To match channel names with a wildcard-style pattern instead, set UseMatchesMask to True on the server. This uses Delphi mask syntax ("*" and "?"), not MQTT syntax, and is applied when the server resolves which live channels a Subscribe, Broadcast or Publish argument expands to. It is most useful on the publish side, to fan a message out to every channel currently matching a pattern:

If you need real MQTT topic filters with "#" and "+" semantics, use the MQTT subprotocol components instead of the sgc subprotocol.

Bindings

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer

Usually, servers have more than one IP. If you enable a WebSocket Server and set the listening port to 80, when the server starts, it tries to listen on port 80 of ALL IPs. So if you have 3 IPs, it will bind port 80 on each of them.

Bindings allow defining which exact IP and Port are used by the Server. Example, if you need to listen on port 80 for IP 127.0.0.1 (internal address) and 80.254.21.11 (public address), you can do this before the server is activated:

```
bind = WSServer->Bindings->Add();
{
  bind->Port = 80;
  bind->IP = "127.0.0.1";
}
bind = WSServer->Bindings->Add();
{
  bind->Port = 80;
  bind->IP = "80.254.21.11";
}
```

Post Big Files

Supported by

TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI

When an HTTP client sends a **multipart/form-data** stream, the stream is saved by server in memory. When the files are big, the server can get an **out of memory** exception. To avoid these exceptions, the server has a property called HTTPUploadFiles where you can configure how the POST streams are handled: in memory or as a file streams. If the streams are handled as file streams, the streams received are stored directly in the hard disk so the memory problems are avoided.

To configure your server to save multipart/form-data streams as file streams, follow the next steps:

1. Set the property **HTTPUploadFiles.StreamType = pstFileStream**. Using this setup, the server will store these streams in the hard disk.
2. You can configure which is the **minimum size in bytes** where the files will be stored as file stream. By default the value is zero, which means all streams will be stored as file stream.
3. The folder where the streams are stored using **SaveDirectory**, if not set, they will be stored in the same folder where the application is.
4. When a client sends a multipart/form-data, the content is encoded inside boundaries, if the property **Remove-Boundaries** is enabled, the content of boundaries will be extracted automatically after the full stream is received.

Sample Code

First create a new server instance and set the Streams are saved as File Streams.

```
TsgcWebSocketHTTPServer *oServer = new TsgcWebSocketHTTPServer();
oServer->Port = 5555;
oServer->HTTPUploadFiles->StreamType = pstFileStream;
oServer->Active = true;
```

Then create a new html file with the following configuration

```
<html>
    <head><title>sgcWebSockets - Upload Big File</title></head>
    <body>
        <form  action="http://127.0.0.1:5555/file"  method="post"  enctype="multipart/
form-data" accept-charset="UTF-8">
            <input type="file" name="file_1" />
            <input type="submit" />
        </form>
    </body>
</html>
```

Finally open the html file with a web browser and send a file to the server. The server will create a new file stream with the extension ".sgc_ps" and when the stream is fully received, it will extract the file from the boundaries.

Events

There are 2 events which can be used to customize the upload file flow (requires the property HTTPUploadFiles.RemoveBoundaries is enabled)

OnHTTPUploadBeforeSaveFile

This event is fired BEFORE the file is saved and allows customizing the name of the file received.

```
void OnHTTPUploadBeforeSaveFileEvent(TObject *Sender, String &aFileName, String &aFilePath)
{
  if (aFileName == "test.jpg")
  {
    aFileName = "custom_test.jpg";
  }
}
```

OnHTTPUploadAfterSaveFile

This event is fired AFTER the file is saved and allows you to know the name of the saved file.

```
void OnHTTPUploadBeforeSaveFileEvent(TObject *Sender, string aFileName, string aFilePath)
{
  DoLog("File Received: " + aFileName);
}
```

OnHTTPUploadReadInput

This event is fired when the decoder reads an input value received different from the file input (example: if the form has some variables like name, date...).

```
void OnHTTPUploadReadInputEvent(TObject *Sender, string aName, string aValue)
{
  DoLog("Input value Received: " + aName + ":" + aValue);
}
```

Compression

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
Web Browsers like Chrome

This is a feature that works very well when you need to send a lot of data, usually using a binary message, because it compresses WebSocket message using protocol "PerMessage_Deflate" which is supported by some browsers like Chrome.
This page is about compressing WebSocket messages. To compress HTTP/1.1 responses (static files and OnCommandGet), see HTTP Compression instead.

To enable this feature, you need to activate the following property:

Extensions/ PerMessage_Deflate / Enabled

When a client tries to connect to a WebSocket Server and this property is enabled, it sends a header with this property enabled. If the server has activated this feature, it sends a response to the client with this protocol activated and all messages will be compressed. If the server does not have this feature, then all messages will be sent without compression.

On Web Browsers, you don't need to do anything, if this extension is supported it will be used automatically, if not, then messages will be sent without compression.

If WebSocket messages are small, it is better not to enable this property because it consumes CPU cycles to compress/decompress messages. But if you are transferring a large amount of data, you will notice an increase in message exchange speed.

HTTP Compression

Supported by
TsgcWebSocketHTTPServer
HTTPCompression compresses HTTP/1.1 responses with gzip or deflate before they are sent to the client, when the client advertises support for it using the **Accept-Encoding** request header. It applies both to static files served from **DocumentRoot** and to responses you build yourself in **OnCommandGet**. It is disabled by default, so existing applications are not affected until you turn it on.
This is a different feature from Compression, which compresses WebSocket messages using the PerMessage_Deflate protocol. HTTPCompression only affects plain HTTP/1.1 requests and responses; it has no effect on WebSocket traffic, and Compression has no effect on HTTP responses.
To enable it, set:
HTTPCompression / Enabled
You can also tune it:
Level — compression level from 0 (none) to 9 (maximum); default 6.
MinSize — responses smaller than this, in bytes, are not compressed; default 1024.
Algorithms — which encodings the server is allowed to use; default both gzip and deflate. When the client accepts both, gzip is preferred.
ContentTypes — list of content types that are eligible for compression, wildcards like "text/*" are supported; defaults to the common text, JSON, JavaScript, XML and SVG types.
The server negotiates the encoding from the request's Accept-Encoding header, honoring a quality value of 0 as a refusal (for example "gzip;q=0"). It never compresses a response that already has a Content-Encoding, a 204/304/1xx status, a Server-Sent Events stream, or a body that would not end up smaller. When it does compress, it sets the response's **Content-Encoding** header and adds **Vary: Accept-Encoding** so caches key on the encoding correctly.
HTTPCompression currently applies to HTTP/1.1 responses only; HTTP/2 responses are not compressed by this property. If you compress content that mixes a secret (such as a CSRF token) with attacker-influenced data over a TLS connection, be aware that response compression is what makes the BREACH class of attack possible; this is a general property of HTTP compression, not something specific to this implementation.

```
oServer = new TsgcWebSocketHTTPServer();
oServer->DocumentRoot = "C:\\inetpub\\wwwroot";
oServer->HTTPCompression->Enabled = true;
oServer->HTTPCompression->MinSize = 1024;
oServer->Active = true;
```

Custom Objects

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI
TsgcWebSocketClient

Every time a new WebSocket connection is established, sgcWebSockets creates a TsgcWSConnection class where you can access some properties like the identifier, bytes received/sent, client IP... and there is a property called **Data** where you can store objects in memory like database access, session objects...

```
// You can create a new class called MyClass and create some properties, example:
class TMyClass
{
private:
  bool FRegistered;
  String FUser;
public:
  __property bool Registered = {read=FRegistered, write=FRegistered};
  __property String User = {read=FUser, write=FUser};
};

// Then, when a new client connects, OnConnect Event, create a new TMyClass and Assign to Data:
void WSServerConnect(TsgcWSConnection *Connection)
{
  Connection->Data = new TMyClass();
}

// Every time a new message is received by the server, you can access your custom object using
// Connection.Data property.
void WSServerMessage(TsgcWSConnection *Connection, string Text)
{
  if (dynamic_cast<TMyClass*>(Connection->Data)->Registered == true)
  {
    DoSomeStuff();
  }
}

// When a connection is closed, you may free your object:
void TfrmServerChat::WSServerDisconnect(TsgcWSConnection *Connection, int Code)
{
  TMyClass *oMyClass = dynamic_cast<TMyClass*>(Connection->Data);
  if (oMyClass != 0)
  {
    delete oMyClass;
    Connection->Data = NULL;
  }
}
```

Groups

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI

sgcWebSockets provides a powerful method for **broadcasting messages to specified subsets of connected clients**. A group can have any number of clients, and a client can be a member of any number of groups. You don't have to explicitly create groups. In effect, a group is automatically created the first time you specify its name in a call to Groups.Add.

When you add a user to a group using the **Groups.Add** method, the user receives messages directed to that group for the duration of the current connection.

Adding and removing users

To add or remove users from a group, you call the Add or Remove methods, and pass the Group Name and the TsgcWSConnection class. You do not need to manually remove a user from a group when the connection ends.

The following example shows the Groups.Add method.

```
void OnConnect(TsgcWSConnection *Connection)
{
  TsgcWebSocketServer1->Groups->Add("Room1", Connection);
}
```

Sending Messages to a Group

You can send a message to all members of a group as shown in the following example.

```
TsgcWebSocketServer1->Groups->Group["Room1"]->Broadcast("Hello Members of Room1");
```

Or you can send a message to all groups that start with "Room" (so if exists Room1, Room2, Room3... these users will receive a message).

```
TsgcWebSocketServer1->Groups->Broadcast("Room*", "Hello Members of Room");
```

Events

There are 2 events that can be used to handle the Groups and Clients every time a new client is added to a group or when one is removed:

OnClientAdded
OnClientRemoved

Example, send a message to the group when a member leaves the group.

```
TsgcWebSocketServer1->Groups->OnClientRemoved() = OnClientRemovedEvent();

void OnClientRemovedEvent(TObject *Sender, const TsgcWSServerGroupItem *aGroup,
  const TsgcWSConnection *aConnection)
{
  aGroup->BroadCast("Client " + aConnection->Guid + " has disconnected");
}
```

IOCP

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer

***Requires custom Indy version.**

IOCP for Windows is an API that allows handling thousands of connections using a limited pool of threads instead of using one thread per connection as Indy does by default.
To enable IOCP for Indy Servers, Go to **IOHandlerOptions** property and select **iohIOCP** as IOHandler Type.

```
Server->IOHandlerOptions->IOHandlerType = iohIOCP;
Server->IOHandlerOptions->IOCP->IOCPThreads = 0;
Server->IOHandlerOptions->IOCP->WorkOpThreads = 0;
Server->IOHandlerOptions->IOCP->PartialRequestTimeout = 0;
Server->IOHandlerOptions->IOCP->TCPKeepAlive->Enabled = true;
Server->IOHandlerOptions->KeepAliveTimeout = 120;
```

IOCPThreads are the threads used for IOCP asynchronous requests (overlapped operations), by default the value is zero which means the number of threads are calculated using the number of processors (except for Delphi 7 and 2007 where the number of threads is set to 32 because the function cpucount is not supported).
WorkOpThreads sets the minimum number of worker threads kept ready to process requests, 0 by default. An IOCP thread only picks up a completed operation and hands the work to this pool, so it never blocks on a slow client. The pool grows on demand up to a fixed maximum whenever every worker is already busy, so it does not need to be sized in advance. Raise this value only if you want the pool already warm before the first request, which avoids paying for thread creation during an initial burst of connections.
Changed in 2026.9.0. WorkOpThreads used to pin every connection to one of these threads, so all requests on a connection ran on the same thread, and **WorkOpRoundRobin** chose how new connections were distributed over them. Pinning has been removed: a connection parked waiting for the rest of a request blocked every other connection assigned to the same thread, which stalled the server. Requests on the same connection can now run on different threads, so anything held per thread, such as a database connection or a thread-local cache, must no longer be assumed to follow a connection. WorkOpRoundRobin no longer exists. Remove it from your code, and if you enabled it in the Object Inspector, delete the WorkOpRoundRobin line from the form's .dfm as well, otherwise the form does not load.
PartialRequestTimeout lets the server wait for the rest of an HTTP request body whose length it already knows, in milliseconds and 0 by default, which disables it. Without it, a request body that does not all arrive within a single completion, for example a large POST sent in several packets, is silently dropped and the connection closed with no reply. The wait never runs on an IOCP thread, which services many connections and must never block, it runs on the worker pool described above, which grows on demand up to a fixed maximum, so a slow or unresponsive client cannot exhaust it and hold back the other connections.
TCPKeepAlive lets the operating system probe idle connections and detect a peer that has silently disappeared (disabled by default). When TCPKeepAlive.Enabled is set to True, the O/S starts probing a connection after it has been idle for TCPKeepAlive.Time seconds (60 by default), repeating every TCPKeepAlive.Interval seconds (10 by default) until the peer answers or the connection is dropped.
KeepAliveTimeout closes any connection that has been idle for longer than the given number of seconds (0 by default, which means disabled). Only idle connections are affected, a request that is being read or a response that is being written is never interrupted by this timeout. This is the closest match to how the default thread-per-connection IOHandler frees a silent connection, and it does not depend on the operating system or on any cooperation from the peer.

Enabling IOCP for windows servers is recommended when you need to handle thousands of connections. If your server is only handling a maximum of 100 concurrent connections, you can stay with the default Indy thread model.

On an SSL binding the IOCP engine does not call a TLS library directly, it drives TLS through a provider, so any TLS library can be plugged in and not only OpenSSL. See TLS Provider.

OnDisconnect event not fired

IOCP works differently from default indy IOHandler. With the default Indy IOHandler, every connection runs in a thread and these threads are running all the time, checking if the connection is active, so if there is a disconnection, it's notified in a short period of time.

IOCP works differently, there is a thread pool which handles all connections, instead of 1 thread = 1 connection like indy does by default. For IOCP, the only way to detect if a connection is still alive is to try writing to the socket. If there is any error, it means that the connection is closed. There are 3 options to detect disconnections:

1. If you use **TsgcWebSocketClient**, you can enable it in Options property, **CleanDisconnect := True** (by default is disabled). If it's enabled, before the client disconnects it sends a message informing the server about disconnection, so the server will receive this message and the OnDisconnect event will be raised.
2. You can enable **heartbeat** on the **server** side, for example every 60 seconds, so it will try to send a ping to all clients connected and if there is any client disconnected, OnDisconnect will be called.
3. You can enable **TCPKeepAlive** and/or set **KeepAliveTimeout** on the server's IOHandlerOptions. These let the server itself detect a peer that has gone silent, for example because the client crashed or the network connection dropped without sending a close message, and close the connection, without depending on the client sending anything.

EPOLL

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer

***Requires sgcWebSockets Enterprise Edition.**

EPOLL for Linux is an API that allows handling thousands of connections using a limited pool of threads instead of using one thread per connection as Indy does by default.
To enable EPOLL for Indy Servers, Go to **IOHandlerOptions** property and select **iohIEPOLL** as IOHandler Type.

```
Server->IOHandlerOptions->IOHandlerType = iohEPOLL;
Server->IOHandlerOptions->EPOLL->EPOLLThreads = 0;
Server->IOHandlerOptions->EPOLL->WorkOpThreads = 0;
Server->IOHandlerOptions->EPOLL->PartialRequestTimeout = 0;
Server->IOHandlerOptions->EPOLL->TCPKeepAlive->Enabled = true;
Server->IOHandlerOptions->KeepAliveTimeout = 120;
```

EPOLLThreads are the threads used for EPOLL asynchronous requests (overlapped operations), by default the value is zero which means the number of threads are calculated using the number of processors (except for Delphi 7 and 2007 where the number of threads is set to 32 because the function cpucount is not supported). You can adjust the number of threads manually.
WorkOpThreads sets the minimum number of worker threads kept ready to process requests, 0 by default. An EPOLL thread only detects that a socket is ready and hands the work to this pool, so it never blocks on a slow client. The pool grows on demand up to a fixed maximum whenever every worker is already busy, so it does not need to be sized in advance. Raise this value only if you want the pool already warm before the first request, which avoids paying for thread creation during an initial burst of connections.
Changed in 2026.9.0. WorkOpThreads used to pin every connection to one of these threads, so all requests on a connection ran on the same thread, and **WorkOpRoundRobin** chose how new connections were distributed over them. Pinning has been removed: a connection parked waiting for the rest of a request blocked every other connection assigned to the same thread, which stalled the server. Requests on the same connection can now run on different threads, so anything held per thread, such as a database connection or a thread-local cache, must no longer be assumed to follow a connection. WorkOpRoundRobin no longer exists. Remove it from your code, and if you enabled it in the Object Inspector, delete the WorkOpRoundRobin line from the form's .dfm as well, otherwise the form does not load.
PartialRequestTimeout lets the server wait for the rest of an HTTP request body whose length it already knows, in milliseconds and 0 by default, which disables it. Without it, a request body that does not all arrive within a single readiness event, for example a large POST sent in several packets, is silently dropped and the connection closed with no reply. The wait never runs on an EPOLL thread, which services many connections and must never block, it runs on the worker pool described above, which grows on demand up to a fixed maximum, so a slow or unresponsive client cannot exhaust it and hold back the other connections.
TCPKeepAlive lets the operating system probe idle connections and detect a peer that has silently disappeared (disabled by default). When TCPKeepAlive.Enabled is set to True, the O/S starts probing a connection after it has been idle for TCPKeepAlive.Time seconds (60 by default), repeating every TCPKeepAlive.Interval seconds (10 by default) until the peer answers or the connection is dropped.
KeepAliveTimeout closes any connection that has been idle for longer than the given number of seconds (0 by default, which means disabled). Only idle connections are affected, a request that is being read or a response that is being written is never interrupted by this timeout. This is the closest match to how the default thread-per-connection IOHandler frees a silent connection, and it does not depend on the operating system or on any cooperation from the peer.

Enabling EPOLL for Linux servers is recommended when you need to handle thousands of connections. If your server is only handling a maximum of 100 concurrent connections, you can stay with the default Indy thread model.

On an SSL binding the EPOLL engine does not call a TLS library directly, it drives TLS through a provider, so any TLS library can be plugged in and not only OpenSSL. See TLS Provider.

OnDisconnect event not fired

EPOLL works differently from default indy IOHandler. With default indy IOHandler, every connection runs in a thread and these thread are running all the time and checking if connection is active, so if there is a disconnection, it's notified in a short period of time.

EPOLL works differently, there is a thread pool which handles all connections, instead of 1 thread = 1 connection like indy does by default. For EPOLL, the only way to detect if a connection is still alive is trying to write in socket, if there is any error means that connection is closed. There are 3 options to detect disconnections:

1. If you use **TsgcWebSocketClient**, you can enable it in Options property, **CleanDisconnect := True** (by default is disabled). If it's enabled, before the client disconnects it sends a message informing the server about disconnection, so the server will receive this message and the OnDisconnect event will be raised.
2. You can enable **heartbeat** on the **server** side, for example every 60 seconds, so it will try to send a ping to all clients connected and if there is any client disconnected, OnDisconnect will be called.
3. You can enable **TCPKeepAlive** and/or set **KeepAliveTimeout** on the server's IOHandlerOptions. These let the server itself detect a peer that has gone silent, for example because the client crashed or the network connection dropped without sending a close message, and close the connection, without depending on the client sending anything.

Linux Connections Limit

If you want to increase the number of concurrent open connections use the following command

ulimit -n 10000

The previous command sets the max number of open files descriptors to 10000

TLS Provider

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcHTTPServer
TsgcWebSocketLoadBalancerServer

***Requires sgcWebSockets Enterprise Edition and the custom Indy version.**

The IOCP and EPOLL server engines do not call a TLS library directly. They drive TLS through a small provider layer, so any TLS library that can encrypt and decrypt over byte buffers can be plugged into them. OpenSSL is the provider that ships with the components and it is still the default, nothing changes for a server that does not replace the SSL handler.
This extension point exists because the engines used to accept only a handler inheriting from the sgc OpenSSL handlers. A handler built on another TLS library, for example one based on SecureBlackbox, could never inherit from those classes, so such a server always fell back to the classic thread per connection engine. With a TLS provider the same server runs on IOCP or EPOLL with its own TLS library.

The two classes

Both classes are declared in **sgcIndy_IOHandler_IO_TLS.pas**.

TsgcTLSProviderContext is created once per listening socket and is shared by every connection accepted on it. It carries the credentials and creates the per connection sessions.

- **CertFile, KeyFile, RootCertFile, Password:** the credentials, filled from the server SSLOptions before Init is called. A provider that does not read certificates from files simply ignores them.
- **ALPNProtocols:** the protocol names offered through the ALPN extension.
- **Init:** called when the server goes active, after the credentials are set. Build the TLS library context here.
- **Shutdown:** called when the server stops. It must leave the context reusable by a later Init, because restarting a server calls Init again on the same object.
- **CreateSession:** returns a new session for one accepted connection.

TsgcTLSProviderSession is created once per connection and holds the TLS state of that connection. It is the crypto core and nothing else.

- **BeginHandShake:** starts the server side handshake.
- **FeedHandShake:** receives one flight of handshake ciphertext and returns tlshsContinue, tlshsDone or tlshsError.
- **Decrypt:** turns received ciphertext into plaintext and returns the plaintext length.
- **Encrypt:** encrypts a range of plaintext and returns the number of plaintext bytes consumed.
- **HandShakeDone:** True once the connection is ready to carry application data.
- **Shutdown:** closes the TLS session and releases whatever the TLS library allocated for it.
- **OnHandShakeData, OnReadData, OnWriteData:** the three events the session uses to hand bytes back to the IOHandler. OnHandShakeData carries ciphertext produced while the handshake is running, OnWriteData carries ciphertext produced by Encrypt, and OnReadData carries plaintext the provider decoded on its own initiative, typically application data that arrived in the same flight that completed the handshake. The IOHandler assigns all three, a provider only raises them.
- **SocketHandle:** the socket descriptor, set before BeginHandShake. Some TLS libraries want it for their own bookkeeping, OpenSSL for instance still asks for it. It is there for that reason only.

The contract

A provider that respects these rules runs on both engines without any further work.

- A session **never** reads from or writes to the socket. The IOHandler owns the socket. Ciphertext that has to go out leaves through the three events, and plaintext comes back from Decrypt.
- **tlshsContinue** returned by FeedHandShake means there is no answer yet and the next flight should be fed in. It is not an error and it does not mean the peer closed.
- **Decrypt works in place.** VBuffer arrives holding ciphertext and is left holding the plaintext. Returning 0 means the ciphertext received so far did not complete a TLS record, which happens whenever a record is split across TCP segments. It does not mean the peer closed, so the engine waits for the rest of the record.

- **Encrypt** returns the number of plaintext bytes it consumed. The ciphertext it produced leaves through OnWriteData.
- **BeginHandShake must be idempotent.** The engine calls it on every task until the handshake reports done, so a second call has to do nothing.
- The engine can read and write the same connection from **different threads**, but never at the same time. Calls into a session are serialized by the IOHandler.

How to plug in a TLS library

1. Descend from **TsgcIdServerIOHandlerTLSIOCP** on Windows or from **TsgcIdServerIOHandlerTLSEPOLL** on Linux. Both are declared in **sgcIndy_IOHandler_IO_Server_TLS.pas** and both descend from TsgcIdServerIOHandlerIOTLS, which already contains all the engine wiring.
2. Override **DoCreateTLSContext** and return your own TsgcTLSProviderContext. That single method is the whole extension point.
3. Assign an instance of your handler in the server OnSSLGetHandler event. The server takes ownership of the handler and frees it.

Nothing else is required. The IOHandlerOptions values, such as threads, timeouts and TCPKeepAlive, are applied to your handler exactly as they are to the OpenSSL one.

Example

A minimal provider, the handler that returns it, and the server wiring. Replace the MyTLSLib calls with the calls of your own TLS library.

And the server wiring.

```
#include "sgcIndy_IOHandler_IO_TLS.hpp"
#include "sgcIndy_IOHandler_IO_Server_TLS.hpp"

class TMyTLSSession : public TsgcTLSProviderSession
{
private:
  bool FStarted;
public:
  virtual void __fastcall BeginHandShake();
  virtual TsgcTLSHandshakeState __fastcall FeedHandShake(const TIdBytes aBuffer);
  virtual int __fastcall Decrypt(TIdBytes &VBuffer);
  virtual int __fastcall Encrypt(const TIdBytes aBuffer, const int aOffset, const int aLength);
  virtual bool __fastcall HandShakeDone();
  virtual void __fastcall Shutdown();
};

class TMyTLSContext : public TsgcTLSProviderContext
{
public:
  virtual void __fastcall Init();
  virtual void __fastcall Shutdown();
  virtual TsgcTLSProviderSession* __fastcall CreateSession();
};

class TMyTLSIOHandler : public TsgcIdServerIOHandlerTLSIOCP
{
protected:
  virtual TsgcTLSProviderContext* __fastcall DoCreateTLSContext();
};

// idempotent, the engine calls it until the handshake is done
void __fastcall TMyTLSSession::BeginHandShake()
{
  if (FStarted)
    return;
  FStarted = true;
  MyTLSLib_AcceptStart(SocketHandle);
}

// hand the received flight to the library, send back what it produced
TsgcTLSHandshakeState __fastcall TMyTLSSession::FeedHandShake(const TIdBytes aBuffer)
{
  TIdBytes vOut = MyTLSLib_HandShake(aBuffer);
  if (vOut.Length > 0)
    DoHandShakeData(vOut);

  if (MyTLSLib_Failed())
    return tlshsError;
  if (MyTLSLib_HandShakeDone())
    return tlshsDone;
  return tlshsContinue; // no answer yet, feed the next flight
```

```
}

// in place: VBuffer arrives with ciphertext and is left with the plaintext.
// 0 means the record is not complete yet, it is not a closed peer.
int __fastcall TMyTLSSession::Decrypt(TIdBytes &VBuffer)
{
  return MyTLSLib_Decrypt(VBuffer);
}

// the ciphertext leaves through OnWriteData, the result is the number of
// plaintext bytes consumed
int __fastcall TMyTLSSession::Encrypt(const TIdBytes aBuffer, const int aOffset,
  const int aLength)
{
  DoWriteData(MyTLSLib_Encrypt(aBuffer, aOffset, aLength));
  return aLength;
}

bool __fastcall TMyTLSSession::HandShakeDone()
{
  return MyTLSLib_HandShakeDone();
}

void __fastcall TMyTLSSession::Shutdown()
{
  MyTLSLib_Close();
}

// CertFile, KeyFile, RootCertFile and Password are already set here
void __fastcall TMyTLSContext::Init()
{
  MyTLSLib_CreateContext(CertFile, KeyFile, Password);
}

// must leave this object ready for a later Init
void __fastcall TMyTLSContext::Shutdown()
{
  MyTLSLib_FreeContext();
}

TsgcTLSProviderSession* __fastcall TMyTLSContext::CreateSession()
{
  return new TMyTLSSession();
}

TsgcTLSProviderContext* __fastcall TMyTLSIOHandler::DoCreateTLSContext()
{
  return new TMyTLSContext();
}

void __fastcall TForm1::FormCreate(TObject *Sender)
{
  sgcWebSocketServer1->SSL = true;
  sgcWebSocketServer1->SSLOptions->CertFile = "certificate.pem";
  sgcWebSocketServer1->SSLOptions->KeyFile = "key.pem";
  sgcWebSocketServer1->IOHandlerOptions->IOHandlerType = iohIOCP;
  sgcWebSocketServer1->OnSSLGetHandler = OnSSLGetHandler;
  sgcWebSocketServer1->Active = true;
}

void __fastcall TForm1::OnSSLGetHandler(TObject *Sender, TwsSSLHandler aType,
  TIdServerIOHandlerSSLBase *&aSSLHandler)
{
  aSSLHandler = new TMyTLSIOHandler(NULL);
}
```

On Linux, descend from **TsgcIdServerIOHandlerTLSEPOLL** instead and set IOHandlerType to **iohEPOLL**. Everything else stays the same, the provider classes are identical on both platforms.

When the handler is not a provider handler

A handler that descends from neither TsgcIdServerIOHandlerTLSIOCP nor TsgcIdServerIOHandlerTLSEPOLL still works. The server detects it when it starts, runs with the classic thread per connection engine instead of the event-driven one, and reports it through the OnError event. Read the **ActiveIOHandlerType** property to know which engine the server is really running.

Native TLS

Supported by

TsgcWebSocketClient
TsgcTCPClient
TsgcHTTP1Client
TsgcHTTP2Client
TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcHTTPServer
TsgcWebSocketProxyServer
TsgcWebSocketLoadBalancerServer

***Requires the sgcCrypto pack, which ships from the Standard edition up.**

The native TLS engine is a TLS 1.3 and TLS 1.2 implementation written entirely in Object Pascal on top of the sgcCrypto units. It is not a binding: there is no OpenSSL DLL to deploy, no SChannel call and no platform TLS library involved. The same source compiles from Delphi 7 to RAD Studio 13 and runs on every platform the components support.

The reason to reach for it is the key exchange. The engine offers the hybrid post-quantum groups of RFC 10024, and **X25519MLKEM768** is the first group it proposes, so a client that switches to it negotiates a post-quantum key exchange against any server that supports one, with no other change to the application. The hybrid groups exist only in TLS 1.3, so a connection that falls back to TLS 1.2 uses a classic curve. See Post-Quantum Cryptography for what those groups are and why a hybrid is the sensible migration today.

Turning it on

Set the IOHandler and nothing else changes. The certificate, key, root certificate, verification and ALPN properties you already use keep working, the engine reads the same ones.

```
// client
oClient = new TsgcWebSocketClient(NULL);
oClient->URL = "wss://www.esegece.com:2053";
oClient->TLS = true;
oClient->TLSOptions->IOHandler = iohNativeTLS;
oClient->TLSOptions->VerifyCertificate = true;
oClient->TLSOptions->RootCertFile = "cacert.pem";
oClient->Active = true;

// server
oServer = new TsgcWebSocketServer(NULL);
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohNativeTLS;
oServer->SSLOptions->CertFile = "certificate.pem";
oServer->SSLOptions->KeyFile = "key.pem";
oServer->Active = true;
```

On a server the engine also runs on the event-driven engines. Set **IOHandlerOptions.IOHandlerType** to iohIOCP on Windows or iohEPOLL on Linux and the handler is built on the matching TLS Provider, so the same configuration works on all three engines.

Selecting **iohNativeTLS** in a build without the sgcCrypto pack raises an exception that says so, rather than falling back to another library silently.

TLS versions

The engine speaks TLS 1.3 and TLS 1.2, as a client and as a server, on the Indy client, on the Indy server and on the IOCP and EPOLL server engines. TLS 1.3 is always chosen when both sides support it. For the native engine, **TLSOptions.Version** on a client and **SSLOptions.Version** on a server is the lowest version allowed.

Version	Negotiates
tlsUndefined (the default)	TLS 1.3 or TLS 1.2
tls1_2	TLS 1.3 or TLS 1.2
tls1_3	TLS 1.3 only
tls1_0, tls1_1	Nothing. A client raises a configuration error when it connects and a server when it starts, because the engine supports TLS 1.2 and TLS 1.3 only.

This is not how the OpenSSL handler reads the property. With **iohOpenSSL**, tls1_2 means exactly TLS 1.2, while with **iohNativeTLS** it means TLS 1.2 or later. The cipher suite list decides the rest. To get TLS 1.2 only, keep Version at tls1_2 and name only TLS 1.2 suites in **NativeTLS_Options.CipherSuites**. A list that names only TLS 1.3 suites gives TLS 1.3 only, and a list that names suites of both versions allows both. Version tls1_3 together with a list of TLS 1.2 suites leaves no version to negotiate, and the connection raises when it starts.

Groups and cipher suites

TLSOptions.NativeTLS_Options on a client, and **SSLOptions.NativeTLS_Options** on a server, carry the two preference lists. Both are strings in the OpenSSL style, items separated by colons, most preferred first, and one list serves TLS 1.3 and TLS 1.2 together. Leave one empty to keep the engine default. An invalid list raises when the connection starts, so a typo is not silently ignored.

Property	Accepted names	Default
Groups	X25519MLKEM768, SecP256r1MLKEM768, SecP384r1MLKEM1024, X25519, secp256r1, secp384r1. The aliases P-256, prime256v1 and P-384 are accepted too, and names are compared without regard to case. The three hybrid groups are TLS 1.3 only.	X25519MLKEM768:X25519:secp256r1:secp384r1
CipherSuites	TLS 1.3: TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256, TLS_AES_128_GCM_SHA256. TLS 1.2: the suites in the table below. IANA names and OpenSSL style names such as ECDHE-RSA-AES128-GCM-SHA256 are both accepted, and the engine reports the IANA name.	TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256:TL plus the TLS 1.2 defaults below

The first group in the list is the one a client sends a key share for, so leaving **X25519MLKEM768** in front is what makes the first flight already post-quantum. Prefixing an item with a question mark skips it when this build does not have it, and any other unknown or unavailable item makes the whole list fail.

TLS 1.2 cipher suites

Only forward secret AEAD suites are enabled by default. The AES-CBC suites and the RSA key exchange are there for peers that cannot do better, and they are offered or accepted only when **CipherSuites** names them.

Suites	Enabled

TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384, TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384, TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256, TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256, TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256, TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	By default, in this order.
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA, TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA, TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA, TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA, TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256, TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384, TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256, TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	Only when named in CipherSuites.
TLS_RSA_WITH_AES_128_GCM_SHA256, TLS_RSA_WITH_AES_256_GCM_SHA384, TLS_RSA_WITH_AES_128_CBC_SHA, TLS_RSA_WITH_AES_256_CBC_SHA, TLS_RSA_WITH_AES_128_CBC_SHA256, TLS_RSA_WITH_AES_256_CBC_SHA256	Only when named in CipherSuites. The RSA key exchange has no forward secrecy.

The ECDHE suites of a TLS 1.2 connection use a classic group of the **Groups** list, X25519, secp256r1 or secp384r1, so keep at least one of them in the list when TLS 1.2 peers must connect. The RSA key exchange uses no group at all.

In a build made with **SGC_CRYPTO_FIPS** the two groups built on X25519, the ChaCha20-Poly1305 suites and the RSA key exchange are compiled out, and the defaults change to **SecP256r1MLKEM768:SecP384r1MLKEM1024:secp256r1:secp384r1** and **TLS_AES_256_GCM_SHA384:TLS_AES_128_GCM_SHA256**, with the four ECDHE AES-GCM suites for TLS 1.2. See FIPS Mode.

TLS 1.2 behaviour

TLS 1.2 is implemented with the protections a modern peer expects, and without the parts of the protocol that caused its known attacks.

- **Extended master secret.** RFC 7627 is offered by a client and accepted by a server. A peer that does not support it is still accepted.
- **No renegotiation.** A renegotiation request is refused with a no_renegotiation alert. Session resumption uses session tickets and requires the extended master secret, see Session resumption below.
- **No SHA-1 signatures.** Following RFC 9155, SHA-1 signatures are neither sent nor accepted. A TLS 1.2 ClientHello without the signature_algorithms extension is refused, because it would mean SHA-1 only.
- **Downgrade protection.** The downgrade protection of RFC 8446 section 4.1.3 works on both sides. A server that also allows TLS 1.3 marks every TLS 1.2 answer, and a client that offered TLS 1.3 refuses a TLS 1.2 answer carrying that mark.
- **Constant time CBC.** CBC records are decrypted in constant time, which closes Lucky Thirteen, and the RSA key exchange applies the countermeasure of RFC 5246 section 7.4.7.1.
- **Certificate key usage.** For the RSA key exchange the server certificate needs the keyEncipherment key usage. The ECDHE suites and TLS 1.3 need digitalSignature.
- **TLS 1.3 only features.** KeyUpdate and exporting keying material exist only on a TLS 1.3 session.

Session resumption

The engine resumes sessions with stateless session tickets, in TLS 1.3 (RFC 8446) and in TLS 1.2 (RFC 5077). After a full handshake the server sends a ticket, sealed with AES-256-GCM under a key that only the server holds. The client keeps it in a session cache, keyed by host, port and ALPN list, and offers it on the next connection, which then skips the certificate exchange. A ticket the server cannot open, or one that has expired, gives a full handshake with a new ticket.

Property	De-fault	Meaning
NativeTLS_Options.SessionTickets	True	A server issues tickets and a client stores and offers them. False turns resumption off, every handshake is then a full one.
NativeTLS_Options.TicketLifetime	7200	Server only. Seconds a ticket can be used to resume, at most 604800. The server ticket key rotates every TicketLifetime seconds, and the previous key stays valid for one more period.

The options are **TLSOptions.NativeTLS_Options** on a client and **SSLOptions.NativeTLS_Options** on a server. The session cache belongs to each client component, so reconnections of the same TsgcWebSocketClient, TsgcHTTP1Client, TsgcTCPClient or TsgcLDAPClient resume, and so does the upstream connection of TsgcWebSocketProxyServer.

TLS 1.2 resumption requires the extended master secret of RFC 7627. A TLS 1.2 session without it is never resumed and gets no ticket, and no ticket is issued either when the server asks for a client certificate. A TLS 1.3 ticket never opens as a TLS 1.2 session, and the other way round.

Whether a connection was resumed is reported by the **Resumed** property of TsgcIdSSLIOHandlerSocketNativeTLS, the handler described below.

What was negotiated

Once the handshake is done, the client handler, **TsgcIdSSLIOHandlerSocketNativeTLS** in the unit sgcSSL_NativeTLS_Indy, reports the result. **VersionName** is TLSv1.2 or TLSv1.3, **CipherSuiteName** is the IANA name of the suite, **GroupName** is the key exchange group, empty for the RSA key exchange, and **PeerSignatureSchemeName** is the signature scheme the server used. The handler is the one the client passes to OnSSLAfterCreateHandler.

Certificates and verification

A server reads **CertFile**, **KeyFile** and **Password**. CertFile and KeyFile can be PEM, or CertFile can be a .pfx or .p12 file, recognised by its extension. A PKCS#12 file holds the certificate, the chain and the private key, so KeyFile is not needed, and it is opened with Password. It is read by the pure Pascal PKCS#12 reader of sgcCrypto_PKCS12, with no OpenSSL. A client that sets **VerifyCertificate** to True checks the server chain against the trusted roots described below and checks that the name matches the host it connected to. **VerifyDepth** is passed through, and a value above 16 is capped at 16.

The trusted roots are the certificates in **RootCertFile**. Setting **NativeTLS_Options.UseSystemRoots** to True adds the certificate authorities the operating system trusts. It is False by default. The two combine: the roots of RootCertFile are loaded first, the system roots are added to the same store, and a chain is accepted when it leads to any of them. When no system root is found, nothing is added and no error is raised, so verification relies on RootCertFile alone. The property is **TLSOptions.NativeTLS_Options.UseSystemRoots** on a client and **SSLOptions.NativeTLS_Options.UseSystemRoots** on a server, where the roots serve only to validate client certificates. Where the system roots come from depends on the platform:

- **Windows.** The ROOT system store, the Trusted Root Certification Authorities, read through crypt32.dll.
- **macOS.** The anchor certificates of Security.framework, with /etc/ssl/cert.pem as the fallback.
- **iOS.** The system roots cannot be listed. When no root is loaded, a chain the engine rejects only because its root is not trusted is evaluated by Security.framework with the SSL policy and the host name. The key usage and host name checks of the engine still run afterwards.
- **Android.** The certificate files of /apex/com.android.conscrypt/cacerts (Android 14 and later) or /system/etc/security/cacerts, PEM or DER. Roots the user disabled, listed in /data/misc/user/0/cacerts-removed, are skipped, and the user roots of /data/misc/user/0/cacerts-added are added when the application can read those folders.
- **Linux and other platforms.** The first of these files that exists and holds at least one certificate: /etc/ssl/certs/ca-certificates.crt, /etc/pki/tls/certs/ca-bundle.crt, /etc/ssl/ca-bundle.pem, /etc/pki/tls/cacert.pem, /etc/ssl/cert.pem and /etc/ssl/certs/ca-bundle.crt.

To load a folder of certificate files yourself, **TsgcTLS13TrustStore.AddDirectory**(aDirectory, aRemovedDirectory) in the unit sgcSSL_NativeTLS_Certificates reads every PEM or DER file of aDirectory, skips any file whose name also exists in aRemovedDirectory, and returns how many certificates it added. It works on every platform.

To pin the name the certificate must carry rather than using the connection host, set **TLSOptions.OpenSSL_Options.X509Checks.HostName** or **IPAddress**, with the matching flag in **X509Checks.Mode**. The engine reads those two values even though it is not OpenSSL, so the property means the same thing whichever IOHandler is selected.

On a server, **SSLOptions.VerifyCertificate** asks the client for a certificate and validates it against the trusted roots described above, and **SSLOptions.VerifyCertificate_Options.FailIfNoCertificate** refuses a client that sends none. **VerifyClientOnce** has nothing to do here. TLS 1.3 has no renegotiation and the engine refuses it in TLS 1.2, so it never asks a second time.

ALPN works as usual through **ALPNProtocols** on a client and through the server ALPN selection, so HTTP/2 over the native engine negotiates **h2** the same way it does over OpenSSL. See ALPN.

QUIC mode

The engine also runs the TLS 1.3 handshake of QUIC and HTTP/3 (RFC 9001). Set **SSLQUICOptions.IOHandler** on TsgcQUICClient, **TLSOptions.IOHandler** on TsgcHTTP3Client, **QUICOptions.SSLOptions.IOHandler** on TsgcQUICServer and TsgcHTTP3Server, or **SSLOptions.IOHandler** on a server with **HTTP3Options.Enabled**, to **iohNativeTLS**.

- The sgc QUIC engine is always used, whatever OpenSSL exports. On a server **QUICMode** reports qmCustom.
- The handshake carries the QUIC transport parameters, requires ALPN and reports TLS alerts as QUIC error codes. The server certificate is read from PEM or PFX, and the peer is verified against RootCertFile or the system roots, host name included.
- Session tickets, resumption and 0-RTT work as on OpenSSL. The server sends a ticket after the handshake, the client stores it with the ALPN and the server transport parameters, and 0-RTT is accepted only once per ticket. QUIC tickets never cross over to TLS over TCP.
- The packet protection is native too, so QUIC and HTTP/3 load no OpenSSL. No define is needed.

See QUIC for the engine comparison.

What it does not do

The engine is deliberately narrow. Everything below is a real limit, not a setting to look for.

- **TLS 1.3 and TLS 1.2 only.** There is no SSL, no TLS 1.0 and no TLS 1.1. A client raises when it connects, and a server when it starts, if **Version** is tls1_0 or tls1_1. A peer that can do neither TLS 1.3 nor TLS 1.2 cannot connect.
- **No PSK without a ticket.** Resumption works only with the session tickets the engine issues itself, see Session resumption above. External pre-shared keys are not supported.
- **No 0-RTT over TCP.** On a TLS connection over TCP early data is neither sent nor accepted. Post-handshake client authentication is not implemented either.
- **QUIC through its own mode.** QUIC and HTTP/3 use the QUIC mode described above, configured on those components.
- **No system roots by default.** Unless **NativeTLS_Options.UseSystemRoots** is True, trust roots come from **RootCertFile** and nowhere else, so a client that verifies certificates must ship or point at a root file. That is the one behaviour difference a working OpenSSL or SChannel configuration is most likely to notice.
- **OnSSLVerifyPeer is not raised.** That event belongs to the OpenSSL handler. The native engine has its own verification hooks, which receive the whole peer chain and the engine verdict at once instead of one certificate per depth, so an application that overrides the trust decision has to use those rather than the component event.

Nothing the peer sends raises an exception. A protocol failure ends the connection with the proper TLS alert, and the reason is reported through the component error events as usual.

The AES-GCM and AES-CBC record protection uses the AES-NI and PCLMULQDQ instructions on Windows builds made with Delphi when the CPU has them, see **sgcAES_HardwareAvailable** in Symmetric Encryption. For a build where no part of the library loads OpenSSL, see Running without OpenSSL.

ALPN

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient

***Requires custom Indy version.**

Application-Layer Protocol Negotiation (ALPN) is a Transport Layer Security (TLS) extension for application-layer protocol negotiation. ALPN allows the application layer to negotiate which protocol should be performed over a secure connection in a manner that avoids additional round trips and which is independent of the application-layer protocols. It is needed by secure HTTP/2 connections, which improves the compresslion of web pages and reduces their latency compared to HTTP/1.x.

Client

You can configure in TLSOptions.ALPNProtocols, which protocols are supported by client. When client connects to server, these protocols are sent on the initial TLS handshake 'Client Hello', and it lists the protocols that the client supports, and server select which protocol will be used, if any.

You can get which protocol has been selected by server accessing to ALPNProtocol property of TsgcWSConnectionClient.

Server

When there is a new TLS connection, OnSSLALPNSelect event is called, here you can access to a list of protocols which are supported by client and server can select which of them is supported.

If there is no support for any protocol, aProtocol can be left empty.

```
// Client
void OnClientConnect(TsgcWSConnection *Connection)
{
  string vProtocol = "";
  vProtocol = dynamic_cast<TsgcWSConnectionClient*>(Connection)->ALPNProtocol;
}

// Server
void OnSSLALPNSelect(TObject *Sender, TStringList *aProtocols, String &aProtocol)
{
  for (int i = 0; i < aProtocols->Count; i++)
  {
    if (aProtocols->Strings[i] == "h2")
    {
      aProtocol = "h2";
      break;
    }
  }
}
```

Forward HTTP Requests

Supported by

TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI
TsgcWSHTTPWebBrokerBridgeServer
TsgcWSHTTP2WebBrokerBridgeServer
TsgcWSServer_HTTPAPI_WebBrokerBridge

You can configure the server to forward some HTTP requests to another server. This is very useful when you have more than one server and only one server is listening on a public address.

Example: you can configure your server, to forward to another server all requests to /internal while all other requests are handled by sgcWebSockets server.

Use the event **OnBeforeForwardHTTP** to check if the URL requested must be forwarded and if it is, then set the URL to forward.

Example: if you want to forward all requests to the document "/internal" to the server "localhost:8080", do the following:

```
void OnBeforeForwardHTTP(TsgcWSConnection *Connection, TIdHTTPRequestInfo *ARequestInfo,
  TsgcWSServerForwardHTTP *aForward)
{
  if (ARequestInfo->Document == "/internal")
  {
    aForward->Enabled = true;
    aForward->URL = "http://localhost:8080";
  }
}
```

Other Options

When you want to forward an HTTP request, you have the following additional options:

1. By default, the request is forwarded using the original document. **Example:** if you forward the request http://localhost:8080/internal to the internal server http://localhost:5555, the forwarded URL will be http://localhost:5555/internal. But you can modify the Document, using the **Document** property of Forward object (by default it will use the same as the original request).

```
aForward.Document = "/NewInternal"
```

2. If you forward a secure HTTP connection (HTTPs), you can customize the SSL/TLS options, in **TLSOptions** property of Forward object. **Example:** set the TLS version

```
aForward.TLSOptions.Version = tls1_2
```

3. The following properties can be used to customize the HTTP request:

- **QueryParams:** the parameters after the document example: 'id=1&user=2'.
- **Host:** specifies the host and port number of the server to which the request is being sent. Example: www.esegece.com:443
- **Origin:** the origin (scheme, hostname, and port) that caused the request. Example: https://www.esegece.com/document.
- **LogFilename:** the name of the filename where the request/response will be stored.
- **NoCache:** if the request must not use the web-browser cache, by default is enabled.
- **CustomHeaders:** a List of custom headers to be added to the request. Example: CustomHeaders.Add('X-ReverseProxy-Host: http://127.0.0.1:8888/test');

Quality Of Service

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc
TsgcWSPClient_MQTT
Java script

SGC Default Protocol and MQTT implement QoS (Quality of Service) for message delivery. There are 3 different types:

Level 0: "At most once", where messages are delivered according to the best efforts of the underlying TCP/IP network. Message loss or duplication can occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.

Level 1: "At least once", where messages are assured to arrive but duplicates may occur.

Level 2: "Exactly once", where messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.

Level 0

The message is delivered according to the best efforts of the underlying TCP/IP network. A response is not expected and no retry semantics are defined in the protocol. The message arrives at the server either once or not at all.

The table below shows the QoS level 0 protocol flow.

Client	Message and direction	Server
QoS = 0	PUBLISH ---------->	**Action:** Publish a message to subscribers

Level 1

The receipt of a message by the server is acknowledged by a ACKNOWLEDGEMENT message. If there is an identified failure of either the communications link or the sending device or the acknowledgement message is not received after a specified period of time, the sender resends the message. The message arrives at the server at least once.

A message with QoS level 1 has a Message ID in the message.

The table below shows the QoS level 1 protocol flow.

Client	Message and direction	Server
QoS = 1 Message ID = x **Action:** Store message	PUBLISH ---------->	**Actions:** • Store message • Publish a message to subscribers • Delete message

Action: Discard message	ACKNOWLEDGEMENT <----------	

If the client does not receive an ACKNOWLEDGMENT message (either within a time period defined in the application, or if a failure is detected and the communications session is restarted), the client may resend the PUBLISH message.

Level 2

Additional protocol flows above QoS level 1 ensure that duplicate messages are not delivered to the receiving application. This is the highest level of delivery, for use when duplicate messages are not acceptable. There is an increase in network traffic, but it is usually acceptable because of the importance of the message content.

A message with QoS level 2 has a Message ID in the message.

The table below shows the QoS level 2 protocol flow. There are two semantics available for how a PUBLISH flow should be handled by the recipient.

Client	Message and direction	Server
QoS = 2 Message ID = x **Action:** Store message	PUBLISH ---------->	**Action:** Store message
	PUBREC <----------	Message ID = x
Message ID = x	PUBREL ---------->	**Actions:** • Publish a message to subscribers • Delete message
Action: Discard message	ACKNOWLEDGEMENT <----------	Message ID = x

If a failure is detected, or after a defined time period, the protocol flow is retried from the last unacknowledged protocol message. The additional protocol flows ensure that the message is delivered to subscribers exactly once.

Queues

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc
Java script

SGC Default Protocol implements Queues to add persistence to published messages (it's only available for **Published messages**)

Level 0: Messages are not queued on Server

Level 1: only last message is queued on Server, and is sent every time a client subscribes to a new channel or connects to the server.

Level 2: All messages are queued on Server, and are sent every time a client subscribes to a new channel or connects to the server.

Level 0

The message is not queued by Server

The table below shows the Queue level 0 protocol flow.

Client	Message and direction	Server
Queue = 0	PUBLISH ---------->	**Action:** Publish a message to subscribers

Level 1

A message with Queue level 1 is stored on the server and if there are other messages stored for this channel, they are deleted.

The table below shows the Queue level 1 protocol flow.

Client	Message and direction	Server
Queue = 1	PUBLISH ---------->	**Actions:** • Deletes All messages of this channel • Store last message by Channel
Action: Process message	NOTIFY <----------	**Action:** Every time a new client subscribes to this channel, the last message is sent.

This is useful where publishers send messages on a "report by exception" basis, where it might be some time between messages. This allows new subscribers to instantly receive data with the retained, or Last Known Good, value.

Level 2

All messages with Queue level 2 are stored on the server.

The table below shows the Queue level 2 protocol flow.

Client	Message and direction	Server
Queue = 2	PUBLISH ---------->	**Action:** Store message
Action: Process message	NOTIFY <----------	**Action:** Every time a new client subscribes to this channel, ALL Messages are sent.

Transactions

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc
Java script

sgcWebSockets SGC Protocol supports transactional messaging, when a client commits a transaction, all messages sent by the client are processed on the server side. There are 3 methods called by the client:

StartTransaction

Creates a New Transaction on the server side and all messages that are sent from the client to the server after this method, are queued on the server side until the client calls Commit or Rollback.

Client	Message and direction	Server
Channel = X	STARTTRANSACTION ---------->	**Action:** Creates a new Queue to store all Messages of the specified channel
Channel = X	PUBLISH ---------->	**Action:** Message is stored on Server Side.
Action: Client gets confirmation of message sent	ACKNOWLEDGEMENT < ----------	**Action:** Server returns an Acknowledgement to the client because message is stored.
....	....	....

Commit

When a client calls Commit, all messages queued by the server are processed.

Client	Message and direction	Server
Channel = X	COMMIT ---------->	**Action:** Process all messages queued by Transaction

RollBack

When a client calls RollBack, all messages queued by the server are deleted and not processed on the server side.

Client	Message and direction	Server
Channel = X	ROLLBACK ---------->	**Action:** Delete all messages queued by Transaction

Message History and Reconnect Recovery

Supported by
TsgcWSPServer_sgc
TsgcWSPClient_sgc
When Message History is enabled on the sgc protocol server, every message published to a channel is assigned a monotonic offset and kept in a bounded per-channel history buffer. A client tracks the last offset it received on each channel and, when it resubscribes after a reconnection, automatically asks the server to replay the messages it missed while it was disconnected.
Message History is opt-in: it is turned off by default, so existing applications behave exactly as before until you enable it.

How it works

Offsets. Each publish to a channel receives a sequential, monotonically increasing offset. The offset identifies the position of a message within the channel history and is the cursor used by clients to request replay.
Per-channel history. The server retains the most recent messages for each channel in a bounded buffer. The buffer is limited by the number of messages (History.Size) and, optionally, by the age of the entries (History.TTLSeconds). Older entries are evicted when either limit is reached.
Client-tracked recovery. The client remembers the last offset it has seen on every channel. After a reconnection, a plain Subscribe sends that last-seen offset to the server, which replays only the messages published after it. The recovered messages are delivered through the same OnEvent / OnMessage handlers as live traffic, so no special handling is required on the client side.

Enabling history (server)

History is configured on the server through the History sub-object. Enable it and set the maximum number of messages retained per channel; optionally set a time-to-live so entries also expire after a number of seconds.

```
sgcWSPServer_sgc1->History->Enabled = true;
sgcWSPServer_sgc1->History->Size = 1000;        // keep last 1000 messages per channel
sgcWSPServer_sgc1->History->TTLSeconds = 3600;  // optional: also expire after 1 hour
```

Reconnect recovery (client)

On the client there is nothing extra to configure. After a reconnection, simply resubscribe to the channels: the client sends its last-seen offset for each channel and the server replays anything missed.

```
// on (re)connect, just resubscribe; the client auto-recovers anything missed
sgcWSPClient_sgc1->Subscribe("news");

// or request the full retained history explicitly (offset 0):
sgcWSPClient_sgc1->Subscribe("news", 0);

// inspect the last offset the client has seen on a channel:
__int64 LastOffset = sgcWSPClient_sgc1->GetLastOffset("news");
```

The single-argument Subscribe call is the recommended pattern, because it recovers exactly the messages the client has not yet seen. Pass an explicit offset only when you want full control over the replay cursor: 0 requests the full retained history, while any other value requests every message published after that offset. Use GetLastOffset to read the cursor the client is currently tracking for a channel.

Configuration reference

The History sub-object on TsgcWSPServer_sgc exposes the following properties.

Property	Type	Default	Description

`History.Enabled`	`Boolean`	`False`	Turns per-channel history on. When False (the default) the server keeps no history and behaves exactly as it did before this feature was introduced.
`History.Size`	`Integer`	`1000`	Maximum number of messages retained per channel. Set to 0 for unlimited (bounded only by available memory and TTLSeconds, if set).
`History.TTLSeconds`	`Integer`	`0`	When greater than 0, entries older than this number of seconds are evicted from the channel history. 0 means no time-based expiry. In .NET this property is named `TtlSeconds` .

Notes

Opt-in. History is disabled by default. Existing behavior is unchanged until History.Enabled is set to True on the server.
Clustering. In a multi-node cluster the history is node-local in this release. Reconnect recovery works when the client reconnects to the same node that holds the channel history. Shared history across cluster nodes is planned for a future release.

TCP Connections

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient

By default, sgcWebSocket uses WebSocket as the protocol, but you can use plain TCP protocol in client and server components.

Client Component
Disable WebSocket protocol.

```
Client->Specifications->RFC6455 = false;
```

Server Component
Handle event OnUnknownProtocol and set Transport as trpTCP and Accept the connection.

```
void OnUnknownProtocol(TsgcWSConnection *Connection, ref bool Accept)
{
  Connection->Transport = trpTCP;
  Accept = true;
}
```

Then when a client connects to the server, this connection will be defined as TCP and will use plain TCP protocol instead of WebSockets. Plain TCP connections do not distinguish between text and binary messages, so all messages received are handled by the OnBinary event.

End of Message
If messages are large, they can sometimes be received fragmented. There is a method to detect the end of a message by specifying which bytes to look for. Example: in the STOMP protocol, all messages end with bytes 0 and 10.

```
void OnWSClientConnect(TsgcWSConnection *Connection)
{
  Connection->TCPEndOfFrameScanBuffer = eofScanAllBytes;
  Connection->AddTCPEndOfFrame(0);
  Connection->AddTCPEndOfFrame(10);
}
```

SubProtocol

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketServer_HTTPAPI

WebSocket provides a simple subprotocol negotiation. It basically adds a header with the protocol names supported by the request. These protocols are received, and if the receiver supports one of them, it sends a response with the supported subprotocol.

sgcWebSockets supports several SubProtocols: MQTT, WAMP... and more. You can implement your own subprotocols using a very easy method, just call RegisterProtocol and send SubProtocol Name as an argument.

Example: you need to connect to a server which implements subprotocol "Test 1.0"

```
Client = new TsgcWebSocketClient();
Client->Host = "server host";
Client->Port = server.port;
Client->RegisterProtocol("Test 1.0");
Client->Active = true;
```

To **use more than 1 protocol in a single connection**, you can use the **Broker Protocol** (Server and Client) components to handle it. Just put a Broker between the Client/Server and the protocols. **Example:** User SGC and Files protocols using a single connection.

```
// ... server
TsgcWebSocketServer *oServer = new TsgcWebSocketServer();
TsgcWSPServer_Broker *oServerBroker = new TsgcWSPServer_Broker();
oServerBroker->Server = oServer;
TsgcWSPServer_sgc *oServerSGC = new TsgcWSPServer_sgc();
oServerSGC->Broker = oServerBroker;
TsgcWSPServer_files *oServerFiles = new TsgcWSPServer_files();
oServerFiles->Broker = oServerBroker;
// ... client
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSPClient_Broker *oClientBroker = new TsgcWSPClient_Broker();
oClientBroker->Client = oClient;
TsgcWSPClient_sgc *oClientSGC = new TsgcWSPClient_sgc();
oClientSGC->Broker = oClientBroker;
TsgcWSPClient_files *oClientFiles = new TsgcWSPClient_files();
oClientFiles->Broker = oClientBroker;
```

When a broker protocol is attached between the Server/Client and the protocol, the events **OnConnect** and **OnDisconnect** are fired in the Broker component (instead of the Server or Client components).

Throttle

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient

Bandwidth Throttling is supported by Server and Client components, if enabled, can limit the number of bits per second sent/received by the socket. Indy uses a blocking method, so if a client is limiting its reading, unread data will be inside the client socket and the server will be blocked from writing new data to the client. The slower the client reads data, the slower the server can write new data.

Server-sent Events (Push Notifications)

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
Java script

SSE are not part of WebSockets, defines an API for opening an HTTP connection for receiving push notifications from a server.

SSEs are sent over traditional HTTP. That means they do not require a special protocol or server implementation to get working. In addition, Server-Sent Events have a variety of features that WebSockets lack by design such as automatic reconnection, event IDs, and the ability to send arbitrary events.

Events

- **Open:** when a new SSE connection is opened.
- **Message:** when the client receives a new message.
- **Error:** when there any connection error like a disconnection.

JavaScript API

To subscribe to an event stream, create an EventSource object and pass it the URL of your stream:

```
var sse = new EventSource('sse.html');

sse.addEventListener('message', function(e)
 {console.log(e.data);
}, false);

sse.addEventListener('open', function(e) {
  // Connection was opened.
}, false);

sse.addEventListener('error', function(e) {
  if (e.readyState == EventSource.CLOSED) {
    // Connection was closed.
  }
}, false);
```

When updates are pushed from the server, the onmessage handler fires and new data is available in its e.data property. If the connection is closed, the browser will automatically reconnect to the source after ~3 seconds (this is a default retry interval, you can change on the server side).

Fields

The following field names are defined by the specification:

event

The event's type. If this is specified, an event will be dispatched on the browser to the listener for the specified event name; the web site would use addEventListener() to listen for named events. the onmessage handler is called if no event name is specified for a message.

data

The data field for the message. When the EventSource receives multiple consecutive lines that begin with data:, it will concatenate them, inserting a newline character between each one. Trailing newlines are removed.

id

The event ID to set the EventSource object's last event ID value to.

retry

The reconnection time to use when attempting to send the event. This must be an integer, specifying the reconnection time in milliseconds. If a non-integer value is specified, the field is ignored.

All other field names are ignored.

For multi-line strings use #10 as line feed.

Examples of use:

If you need to send a message to a client, just use WriteData method.

```
// If you need to send a message to a client, just use WriteData method.
Connection->WriteData("Notification from server");

// To send a message to all Clients, use Broadcast method.
Connection->Broadcast("Notification from server");

// To send a message to all Clients using url 'sse.html', use Broadcast method and Channel parameter:
Connection->Broadcast("Notification from server", "/sse.html");

// You can send a unique id with an stream event by including a line starting with "id:":
Connection->WriteData("id: 1 \r data: Notification from server");

// If you need to specify an event name:
Connection->WriteData("event: notifications \r data: Notification from server");
```

javascript code to listen "notifications" channel:

```
   sse.addEventListener('notifications',      function(e) {
      console.log('notifications:'  + e.data);
   },  false);
```

LoadBalancing

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketLoadBalancerServer

Load Balancing allows distributing work between several back-end servers, every time a new client requests a connection, it connects to a load balancer server (which is connected to back-end servers) and returns a connection string with information about the host, port... which is used by the client to connect to a server. If you have for example 4 servers, with this method all servers will have, more or less, the same number of connections, and workload will be similar.

If a client wants to send a message to all clients of all servers, just use the Broadcast method, and the message will be broadcast to all servers connected to the Load Balancer Server.

To enable this feature:

1. Drop a TsgcWebSocketLoadBalancerServer component, set a listening port and set active to True.

2. Server and Client components have a property called LoadBalancer, where you need to set the host and port of the Load Balancer Server and set Enabled to True.

The component allows load balancing of **WebSocket** and **HTTP** protocols.

Files

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc

This protocol allows sending files from client to server and from server to client in an easy way. You can send from really small files to big files using a low memory usage. You can set:

1. Packet size in bytes.
2. Use custom channels to send files to only subscribed clients.
3. The progress of file send and received.
4. Authorization of files received.
5. Acknowledgement of packets sent.

Proxy

Supported by

TsgcWebSocketClient

Client WebSocket components support WebSocket connections through HTTP proxies. To enable proxy connection, you need to activate the following properties:

Proxy / Enabled

Once set to True, you can set up:

Host: Proxy server address
Port: Proxy server port
UserName/Password: Authentication to connect to proxy, only if required.
ProxyType: the following proxies are supported:

- HTTP
- Socks4
- Socks4A
- Socks5

You can configure SOCKS proxies by accessing the SOCKS property and setting Enabled to True.

Fragmented Messages

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketServer_HTTPAPI

By default, when a stream is sent using sgcWebSockets library, it sends all data in a single packet or buffers all packets and when the latest packet is received, OnBinary message event is called.
This behavior can be customized by the **Options.FragmentedMessages** property, which accepts following values:

1. frgOnlyBuffer: this is the default value. It means that packet messages will be buffered and only when the entire stream is received will the OnBinary message event be called.
2. frgOnlyFragmented: this means that OnFragmented event only will be called for every packet received.
3. frgAll: this means that OnFragmented event will be called for every packet received and when the full stream is received.

OnFragmented event is useful when you must send large streams and the receiver must show progress of the transfer.

Example: the client must send a stream of size 1.000.000 bytes to server and the server wants to show progress for every 1000 bytes received.

The client will send a stream using writedata method with a size for a packet of 1000

```
Client->WriteData(stream, 1000);
```

The server will set in Options.FragmentedMessages := frgAll and will handle OnFragmented event to receive progress of streams

```
void OnFragmented(TsgcWSConnection *Connection, const TMemoryStream *Data, const TOpCode OpCode, const boolean Co
{
  ShowProgress(Data->Size);
  if (Continuation == false)
  {
    SaveStream(Data);
  }
}
```

TsgcWebSocketClient

Client WebSocket component — connects to any RFC 6455 WebSocket server and sends/receives text and binary messages.

Introduction

TsgcWebSocketClient implements Client WebSocket Component and can connect to a WebSocket Server. Follow the steps below to configure this component:
1. Drop a **TsgcWebSocketClient** component onto the form.
2. Set **Host** and **Port** (default is 80) to connect to an available WebSocket Server. You can set **URL** property and Host, Port, Parameters... will be updated from URL. **Example:** wss://127.0.0.1:8080/ws/ will result in:

```
oClient = new TsgcWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClient->TLS = true;
oClient->Options->Parameters = "/ws/";
```

3. You can select if you require **TLS** (secure connection) or not, by default, it is not activated.
4. You can connect through an HTTP Proxy Server, you need to define proxy properties:
Host: proxy server hostname.
Port: proxy server port number.
Username: username for authentication, leave blank for anonymous.
Password: password for authentication, leave blank for anonymous.
5. If the server supports **compression**, you can enable compression to compress messages that are sent.
6. Set **Specifications** allowed, by default, all specifications are enabled.
RFC6455: is standard and recommended WebSocket specification.
Hixie76: always is false
7. Set `Active = true` to start a new WebSocket connection.

Reference

- Properties — 23 properties available on the component.
- Methods — 22 public methods.
- Events — 16 events you can handle.

Guides

Most common uses

- **Connection**
  - **How to Connect to a WebSocket Server**
  - **Open a Client Connection**
  - **Close a Client Connection**
  - **Keep Connection active**
  - **Dropped Disconnections**
  - **Connect TCP Server**
  - **WebSocket Redirections**
- **Secure Servers**
  - **Connect Secure Server**
  - **Certificates OpenSSL**
  - **Certificates SChannel**

- **SChannel Get Connection Info**
- **Send Messages**
  - **Send Text Message**
  - **Send Binary Message**
- **Receive Messages**
  - Receive Text Messages
  - Receive Binary Messages
- **Authentication**
  - **Client Authentication**
- **Other**
  - **Client Exceptions**
  - **Client WebSocket HandShake**
  - **Client Register Protocol**
  - **Client Proxies**

TsgcWebSocketClient | Connect WebSocket Server

URL Property

The easiest way to connect to a WebSocket server is to use the **URL** property and set **Active = true**.

Example: connect to www.esegece.com using secure connection.

```
oClient = new TsgcWebSocketClient();
oClient->URL = "wss://www.esegece.com:2053";
oClient->Active = true;
```

Host, Port and Parameters

You can connect to a WebSocket server using Host and port properties.

Example: connect to www.esegece.com using secure connections

```
oClient = new TsgcWebSocketClient();
oClient->Host = "www.esegece.com";
oClient->Port = 2053;
oClient->TLS = true;
oClient->Active = true;
```

TsgcWebSocketClient | Client Open Connection

Once your client is configured to connect to a server, there are 3 different options to open a new connection.

Active Property

The easiest way to open a new connection is to set the Active property to true. This will attempt to connect to the server using the component configuration.
If you set the Active property to false, it will close the connection if active.
This method is executed in the same thread as the caller. So if you call it from the Main Thread, the method will be executed in the Main Thread of the application.

Open Connection

```
oClient = new TsgcWebSocketClient();
...
oClient->Active = true;
```

When you call Active = true, **you still cannot send any data to the server** because the client may still be connecting. You must first wait until the OnConnect event is fired, and then you can start sending messages to the server.

Close Connection

```
oClient->Active = false;
```

When you call Active = false, **you cannot be sure that connection is already closed** just after this code, so you must wait until the OnDisconnect event is fired.

Start/Stop methods

When you call Start() or Stop() to connect/disconnect from the server, the call is executed in a secondary thread, so it does not block the thread where it is called. Use this method if you want to connect to a server and let your code below continue.

Open Connection

```
oClient = new TsgcWebSocketClient();
...
oClient->Start();
```

When you call Start(), **you still cannot send any data to the server** because the client may still be connecting. You must first wait until the OnConnect event is fired, and then you can start sending messages to the server.

Close Connection

```
oClient->Stop();
```

When you call Stop(), **you cannot be sure that connection is already closed** just after this code, so you must wait until the OnDisconnect event is fired.

Connect/Disconnect methods

When you call Connect() or Disconnect() to open/close a connection to the server, the call is executed in the same thread where it is called, but it waits until the process is finished. You must set a Timeout to define the maximum time to wait until the process is finished (by default 10 seconds).

Example: connect to server and wait up to 5 seconds

```
oClient = new TsgcWebSocketClient();
...
if (oClient->Connect(5000) == true)
{
  oClient->WriteData("Hello from client");
}
else
{
  Error();
}
```

If the Connect() method returns a successful result, you can already send a message to the server because the connection is alive.

Example: disconnect from server and wait up to 10 seconds

```
if (oClient->Disconnect(10000) == true)
{
  ShowMessage("Disconnected");
}
else
{
  ShowMessage("Not Disconnected");
}
```

If the Disconnect() method returns a successful result, this means that the connection is already closed.

OnBeforeConnect event can be used to customize the server connection properties before the client tries to connect to it.

TsgcWebSocketClient | Client Close Connection

Connection can be closed using Active property, Stop or Disconnect methods, read more from Client Open Connection.

CleanDisconnect

When a connection is closed, you can notify the other peer that the connection is being closed by sending a close message. To enable this feature, set the Options.CleanDisconnect property to true.
If this property is enabled, before the connection is closed, a Close message will be sent to the server to notify that the client is closing the connection.

Disconnect

TsgcWSConnection has a method called Disconnect(), that allows you to disconnect the connection at the socket level. If you call this method, the socket will be disconnected directly without waiting for any response from the server. You can send a Close Code with this method.

Close

TsgcWSConnection has a method called Close(), which allows you to send a message to the server requesting to close the connection. If the server receives this message, it must close the connection and the client will receive a notification that the connection is closed. You can send a Close Code with this method.

TsgcWebSocketClient | Client Keep Connection Open

Once your client has connected to a server, sometimes the connection can be closed due to poor signal, connection errors, etc. There are 2 properties that help keep the connection active.

HeartBeat

HeartBeat property allows you to **send a Ping** every X **seconds** to **keep the connection alive**. Some servers close TCP connections if there is no data exchanged between peers. HeartBeat solves this problem by sending a ping at a specific interval. Usually this is enough to maintain a connection active, but you can set a TimeOut interval if you want to close the connection when a response from the server is not received after X seconds.

Example: send a ping every 30 seconds

```
oClient = new TsgcWebSocketClient();
oClient->HeartBeat->Interval = 30;
oClient->HeartBeat->Timeout = 0;
oClient->HeartBeat->Enabled = true;
oClient->Active = true;
```

There is an event called **OnBeforeHeartBeat** which allows customizing HeartBeat behavior. By default, if HeartBeat is enabled, the client will send a WebSocket ping every X seconds as set by the HeartBeat.Interval property.
OnBeforeHeartBeat has a parameter called Handled, by default is false, which means the flow is controlled by TsgcWebSocketClient component. If you set the value to True, then ping won't be sent, and you can send your custom message using Connection class.

WatchDog

If WatchDog is enabled, when the client detects a disconnection, WatchDog tries to reconnect every X seconds until the connection is active again.

Example: reconnect every 10 seconds after a disconnection with unlimited attempts.

```
oClient = new TsgcWebSocketClient();
oClient->WatchDog->Interval = 10;
oClient->WatchDog->Attempts = 0;
oClient->WatchDog->Enabled = true;
oClient->Active = true;
```

You can use **OnBeforeWatchDog** event to change the server where the client will attempt to connect. **Example:** after 3 retries, if the client cannot connect to a server, it will attempt to connect to a secondary server.
The **Handled** property, if set to True, means that the client won't try to reconnect.

TsgcWebSocketClient | Dropped Disconnections

Once the connection has been established, if no peer sends any data, then no packets are sent over the net. TCP is an idle protocol, so it assumes the connection is still active.

Disconnection reasons

- **Application closes:** when a process is finished, usually sends a FIN packet which acknowledges to the other peer that the connection has been closed. But if a process crashes, there is no guarantee that this packet will be sent to the other peer.
- **Device Closes:** if the device closes, most probably there will be no notification about this.
- **Network cable unplugged:** if the network cable is unplugged it is the same as a router closing; there is no data being transferred so the connection is not closed.
- **Loss of signal from router:** if the application loses signal from the router, the connection will still be alive.

Detect Half-Open Disconnections

You can try to detect disconnections using the following methods

Second Connection

You can try to open a second connection and attempt to connect, but this has some disadvantages: you are consuming more resources, creating new threads, etc. Also, if the other peer has rebooted, the second connection will work but the first will not.

Ping other peer

If you try to send a ping or whatever message with a half-open connection, you will see that you don't get any error.

Enable KeepAlive at TCP Socket level

A TCP keep-alive packet is simply an ACK with the sequence number set to one less than the current sequence number for the connection. A host receiving one of these ACKs responds with an ACK for the current sequence number. Keep-alives can be used to verify that the computer at the remote end of a connection is still available. TCP keep-alives can be sent once every TCPKeepAlive.Time (defaults to 7,200,000 milliseconds or two hours) if no other data or higher-level keep-alives have been carried over the TCP connection. If there is no response to a keep-alive, it is repeated once every TCPKeepAlive.Interval seconds. KeepAliveInterval defaults to 1000 milliseconds.

You can enable per-connection KeepAlive and allow the TCP protocol to check if the connection is active or not. This is the preferred method if you want to detect dropped disconnections (for example: when you unplug a network cable).

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
oClient->TCPKeepAlive->Enabled = true;
oClient->TCPKeepAlive->Time = 5000;
oClient->TCPKeepAlive->Interval = 1000;
```

TsgcWebSocketClient | Connect TCP Server

TsgcWebSocketClient can connect to WebSocket servers, but it can also connect to plain TCP Servers.

URL Property

The easiest way to connect to a TCP server is to use the **URL** property and set **Active = true**.

Example: connect to 127.0.0.1 port 5555

```
oClient = new TsgcWebSocketClient();
oClient->URL = "tcp://127.0.0.1:5555";
oClient->Active = true;
```

Host, Port and Parameters

You can connect to a TCP server using Host and port properties.

Example: connect to 127.0.0.1 port 5555

```
oClient = new TsgcWebSocketClient();
oClient->Specifications->RFC6455 = false;
oClient->Host = "127.0.0.1";
oClient->Port = 5555;
oClient->Active = true;
```

TsgcWebSocketClient | Connections TIME_WAIT

When a client initiates a disconnection from the server, there is an exchange between client and server to communicate the state of the disconnection. When the process is finished, the client socket connection enters the TIME_WAIT state for a variable duration. This is normal behavior; in Windows operating systems, this time defaults to about 4 minutes.

You can reduce or eliminate this behavior, but do so with caution, using the following alternatives.

REGEDIT

You can reduce the TIME_WAIT value using the Windows Regedit

1. Open Regedit and access to HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TCPIP\Parameters registry subkeys.
2. Create a new REG_DWORD value named **TcpTimedWaitDelay**
3. Set the value in seconds. Example: if you set a value of 5, it means that TIME_WAIT will wait for a maximum of 5 seconds.
4. Save and restart the system.

LINGER

Another option to avoid the TIME_WAIT state is to use the socket option SO_LINGER. If enabled, instead of closing the connection gracefully, the client resets the connection so the TIME_WAIT state is avoided.
You can enable this option using the **LingerState** property, which by default has a value of -1. If you set a value of zero, the connection will be reset when disconnecting from the socket without a timeout.
This option is probably the least recommended and should only be used as a last resort.

TsgcWebSocketClient | WebSocket Redirections

When the client connects to a WebSocket server, the server can return an HTTP Response Code 30x. If the response code is 301, it means that the location has been moved permanently, and the new URL is provided in the Location HTTP Header.

The WebSocket client handles redirections automatically, so if it detects that the server response contains a redirection, it will disconnect the current connection and attempt to connect to the new Location URL.

Example

1. Client first tries to connect to url ws://127.0.0.1:5000
2. Server returns a Response Code of 301 and contains a Header Location with the value ws://80.50.1.2:3000
3. Client reads the Response from server, detects that it is a redirection and reads the Location
    1. First disconnects the current connection.
    2. Updates the URL property with the value of the Location Header (ws://80.50.1.2:3000)
    3. Connects to the new server.

TsgcWebSocketClient | Connect Secure Server

TsgcWebSocketClient can connect to WebSocket servers using secure and non-secure connections.

You can configure a secure connection, using URL property or Host / Port properties, see Connect to WebSocket Server.

TLSOptions

In **TLSOptions** property there are the properties to **customize a secure connection**. The most important property is **version**, which specifies the **version of TLS protocol**. Usually setting **TLS property to true** and **TLSOptions.Version to tlsUndefined** is enough for the wide majority of WebSocket Servers.

TLSOptions.Version allows you to set the TLS version used to connect to server or let the client negotiate the TLS version from all available (this is the default when value is **tlsUndefined**).

If you get an **error trying to connect to a server** about TLS protocol, **most probably** the server r**equires a newer TLS version** than what you have set.

If **TLSOptions.IOHandler** is set to **iohOpenSSL**, you need to **deploy OpenSSL libraries** (which are the libraries that handle all TLS stuff), check the following article about OpenSSL.
If **TLSOptions.IOHandler** is set to **iohSChannel**, then there is **no need to deploy** any library (only Windows is supported).

TsgcWebSocketClient | Certificates OpenSSL

When the server requires that the client connects using an SSL Certificate, use the TLSOptions property of TsgcWebSocketClient to set the certificate files. The certificate must be in PEM format, so if the certificate has a different format, it must first be converted to PEM.

Connection through OpenSSL libraries requires that TLSOptions.IOHandler = iohOpenSSL.

Configure the following properties:

- **CertFile:** is the path to the certificate in PEM format.
- **KeyFile:** is the path to the private key of the certificate.
- **RootCertFile:** is the path to the root of the certificate.
- **Password:** if the certificate is protected by a password, set the secret here.

TsgcWebSocketClient | Certificates SChannel

When the server requires that the client connects using an SSL Certificate, use the TLSOptions property of TsgcWebSocketClient to set the certificate files.

Connection through SChannel requires that TLSOptions.IOHandler = iohSChannel.

SChannel supports 2 types of certificate authentication:

1. Using a **PFX certificate**
2. Setting the **Hash Certificate** of an already installed certificate in the windows system.

PFX Certificate

PFX Certificate is a file that contains the certificate and private key, sometimes you have a certificate in PEM format, so before using it you must convert it to PFX.
Use the following openssl command to convert a PEM certificate to PFX

```
openssl pkcs12 -inkey certificate-pem.key -in certificate-pem.crt -export -out certificate.pfx
```

Once the certificate is in PFX format, you only need to deploy the certificate and set the **TLSOptions.CertFile** property to its path.

```
TLSOptions.IOHandler = iohSChannel
TLSOptions.CertFile = <certificate path>
TLSOptions.Password = <certificate optional password>
```

Hash Certificate

If the certificate is already installed in the Windows certificate store, you only need to know the certificate thumbprint and set it in the TLSOptions.SChannel_Options property.

Finding the hash of a certificate is as easy in **powershell** as running a **dir** command on the certificates container.

```
dir cert:\localmachine\my
```

The hash is the hexadecimal **Thumbprint** value.

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint                                Subject
----------                                -------
C12A8FC8AE668F866B48F23E753C93D357E9BE10  CN=*.mydomain.com
```

Once you have the Thumbprint value, you must set the hash and the certificate location in the **TLSOptions.SChannel_Options** property.

```
TLSOptions.IOHandler = iohSChannel
TLSOptions.SChannel_Options.CertHash = <certificate thumbprint>
TLSOptions.SChannel_Options.CertStoreName = <certificate store name>
```

```
TLSOptions.SChannel_Options.CertStorePath = <certificate store path>
TLSOptions.Password = <certificate optional password>
```

TsgcWebSocketClient | SChannel Certificate Revocation

Setting **TLSOptions.VerifyCertificate** to True establishes that the server certificate chain is trusted and that it names the host being connected to. It says nothing about whether that certificate has been revoked since it was issued.
That gap matters: a stolen private key whose certificate has been revoked but has not yet expired still authenticates successfully, which is precisely the case revocation exists to cover. Revocation is checked only when it is asked for, through **TLSOptions.SChannel_Options.Revocation**.

```
oClient.TLS := True;
oClient.TLSOptions.IOHandler := iohSChannel;
oClient.TLSOptions.VerifyCertificate := True;
oClient.TLSOptions.SChannel_Options.Revocation.Check := scrcChainExcludeRoot;
```

The status is retrieved while the certificate chain is built, which means the CRL distribution point or OCSP responder named by the certificate is contacted during the handshake. That is why the check is off by default: enabling it adds network traffic to every new connection.

Check

Selects how much of the chain has its revocation status looked up.

scrcDisabled	Default. No revocation status is retrieved and none is required.
scrcEndCertificate	Only the server certificate is checked. The cheapest option, one lookup per connection.
scrcChainExcludeRoot	The server certificate and the intermediate certificates are checked, the root is not. The usual choice, because a self-signed root rarely publishes a CRL for itself.
scrcChain	The whole chain including the root.

Timeout

Milliseconds allowed for the whole revocation retrieval, not per URL, so a responder that does not answer cannot stall the handshake. Default 5000. A value of zero leaves the Windows default in force, which can be considerably longer.

CacheOnly

When True, only revocation data Windows has already cached is used and the handshake never goes to the network, so it is never delayed. A cold cache then leaves the status unknown, which **IgnoreNoRevocationCheck** decides. Default False.

IgnoreRevocationOffline

Decides what happens when the responder could not be reached. Default True, accept the certificate, which is the soft-fail behaviour web browsers use. Set it to False to reject instead. Rejecting is stricter, but it makes every connection depend on the responder being reachable, so a responder outage becomes an outage of your application.

IgnoreNoRevocationCheck

Decides what happens when the certificate publishes no CRL distribution point and no OCSP responder at all, so its status cannot be established. Default True, accept it.
Neither Ignore property ever accepts a certificate that came back **confirmed revoked**. That outcome is always rejected.

Reported errors

When the connection is refused, the reason reaches the OnSChannelVerifyPeer event as one of:

ecCryptRevoked	The certificate has been revoked by its issuer. Always rejected.
ecCryptRevocationOffline	The responder could not be reached. Only rejected when IgnoreRevocationOffline is False.
ecCryptNoRevocationCheck	The certificate publishes no revocation information. Only rejected when IgnoreNoRevocationCheck is False.

Secure defaults

Setting **TLSOptions.Preset** to **tlspSecureDefaults** enables the check as **scrcChainExcludeRoot**, together with the other hardened defaults, and keeps the two soft-fail settings, so switching it on does not break a connection that used to work. A scope you chose yourself before applying the preset is left untouched.

```
oClient.TLSOptions.Preset := tlspSecureDefaults;
```

TsgcWebSocketClient | SChannel Get Connection Info

Once the client has connected to the secure server, you can request information about which TLS version is being used (TLS 1.2, TLS 1.3, etc.), the cipher used, strength, and more.

Call the function **GetInfo** of the SChannel Handler to access this info. You can access the SSL Handler using the method **OnSSLAfterCreateHandler**, which is called after the SChannel Handler is created. After the client connects to the server, if the SSL Handler is assigned, call the function **GetInfo** and if successful, will return the connection data.

```
oClient = new TsgcWebSocketClient();
oClient->URL = "wss://www.esegece.com:2053";
oClient->TLSOptions->Version = tls1_2;
oClient->TLSOptions->IOHandler = iohSChannel;
oClient->OnSSLAfterCreateHandler = OnSSLAfterCreateHandlerEvent;
oClient->OnConnect = OnConnectEvent;
oClient->Active = true;
void OnSSLAfterCreateHandlerEvent(TObject *Sender, TwsSSLHandler aType,
  TIdSSLIOHandlerSocketBase *aSSLHandler)
{
  if (aSSLHandler->ClassType() == __classid(TsgcIdSSLIOHandlerSocketSChannel))
  {
    SSL = dynamic_cast<TsgcIdSSLIOHandlerSocketSChannel*>(aSSLHandler);
  }
}
void OnConnectEvent(TsgcWSConnection *Connection)
{
  if (SSL != NULL)
  {
    TsgcSChannelConnectionInfo oInfo = SSL->GetInfo();
    if (oInfo->Protocol != tls1_2)
    {
      throw Exception("Client cannot connect using TLS 1.2");
    }
  }
}
```

TsgcWebSocketClient | Client Send Text Message

Once the client has connected to the server, it can send text messages. To send a text message, just call the Write-Data() method.

Send a Text Message

Call the **WriteData()** method to send a text message. This method is executed on the **same thread** from which it is called.

```
TsgcWebSocketClient1->WriteData("My First sgcWebSockets Message!.");
```

If **QueueOptions.Text** has a **different value from qmNone**, instead of being processed on the same thread that is called, it will be processed on a secondary thread. By default this option is disabled.

Send a Text Message and Wait for the Response

Call the WriteAndWaitData() method to send a text message and wait for a response from the server. The function returns the text message received.

```
TsgcWebSocketClient1->WriteAndWaitData("My First sgcWebSockets Message!.");
```

TsgcWebSocketClient | Client Send Binary Message

Once the client has connected to the server, it can send binary messages. To send a binary message, just call the WriteData() method.

Send a Binary Message

Call the **WriteData()** method to send a binary message. This method is executed on the **same thread** from which it is called.

```
TMemoryStream oStream = new TMemoryStream();
...
TsgcWebSocketClient1->WriteData(oStream);
delete oStream;
```

If **QueueOptions.Binary** has a **different value from qmNone**, instead of being processed on the same thread that is called, it will be processed on a secondary thread. By default this option is disabled.

Send a Binary Message and Wait for the Response

Call the WriteAndWaitData() method to send a binary message and wait for a response from the server. The function returns the binary message received.

```
TsgcWebSocketClient1->WriteAndWaitDataData(oStream);
```

TsgcWebSocketClient | Client Send a Text and Binary Message

WebSocket protocol only allows two types of messages: Text or Binary. However, you cannot send binary data along with text in the same message.
One way to solve this is to add a header to the binary message before it is sent and decode this binary message when it is received.

There are 2 functions in sgcWebSocket_Helpers which can be used to set a short description of a binary packet. This basically adds a header to the stream which is used to identify the binary packet.

Before sending a binary message, call the method to encode the stream.

```
sgcWSStreamWrite("00001", oStream);
TsgcWebSocketClient1->WriteData(oStream);
```

When a binary message is received, call the method to decode the stream.

```
sgcWSStreamRead(oStream, vID);
```

The only limitation is that the text used to identify the binary message has a maximum length of 10 characters (this can be modified if you have access to source code).

TsgcWebSocketClient | Receive Text Messages

When the client receives a Text Message, the **OnMessage** event is fired. Read the Text parameter to retrieve the string of the message received.

```
void OnMessage(TsgcWSConnection *Connection, const string Text)
{
  ShowMessage("Message Received from Server: " + Text);
}
```

By default, client uses **neAsynchronous** method to dispatch OnMessage event, this means that **this event is executed on the context of Main Thread**, so it's thread-safe to update any control of a form for example.

If your client receives lots of messages or you need to control the synchronization with other threads, set NotifyEvents property to **neNoSync**, this means that OnMessage event will be e**xecuted on the context of connection thread**, so if you require to update any control of a form or access shared objects, you must implement your own synchronization methods.

TsgcWebSocketClient | Receive Binary Messages

When the client receives a Binary Message, the **OnBinary** event is fired. Read the Data parameter to retrieve the binary message received.

```
void OnBinary(TsgcWSConnection *Connection, const TMemoryStream *Data)
{
  oBitmap = new TBitmap();
  oBitmap->LoadFromStream(Data);
  Image1->Picture->Assign(oBitmap);
  Log(
      "#image uncompressed size: " + IntToStr(Data->Size) +
      ". Total received: " + IntToStr(Connection->RecBytes));
  delete oBitmap;
}
```

By default, client uses **neAsynchronous** method to dispatch OnMessage event, this means that **this event is executed on the context of Main Thread**, so it's thread-safe to update any control of a form for example.

If your client receives lots of messages or you need to control the synchronization with other threads, set NotifyEvents property to **neNoSync**, this means that OnMessage event will be e**xecuted on the context of connection thread**, so if you require to update any control of a form or access shared objects, you must implement your own synchronization methods.

TsgcWebSocketClient | Client Authentication

TsgcWebSocket client supports 4 types of Authentications:

- **Basic:** sends an HTTP Header during WebSocket HandShake with User and Password encoded as Basic Authentication.
- **Token:** sends a Token as an HTTP Header during the WebSocket HandShake. Set the required token in Authentication.Token.AuthToken as required by the server.
- **Session:** first the client requests an HTTP session from the server and if the server returns a session, it is passed in the GET HTTP Header of the WebSocket HandShake. (* own Authentication method for sgcWebSockets library).
- **URL:** the client requests authorization using the GET HTTP Header of the WebSocket HandShake. (* own Authentication method for sgcWebSockets library).

Authentication Basic

This is a simple authorization method where user and password are encoded and passed as an HTTP Header. Just set the User and Password and enable only the Basic Authorization type to use this method.

```
oClient = new TsgcWebSocketClient();
oClient->
Authentication
->Enabled = true;
oClient->
Authentication
->Basic->Enabled = true;
oClient->
Authentication
->User = "your user";
oClient->
Authentication
->Password = "your password";
oClient->
Authentication
->Token->Enabled = false;
oClient->
Authentication
->URL->Enabled = false;
oClient->
Authentication
->Session->Enabled = false;
oClient->Active = true;
```

Authentication Token

Allows you to authorize using JWT. This requires you to obtain a token using an external tool (for example: an HTTP connection, OAuth2, etc.).
If you attach an OAuth2 component, you can obtain this token automatically. Read more about OAuth2.
You must set your AuthToken and enable Token Authentication.

```
oClient = new TsgcWebSocketClient();
oClient->
Authentication
```

```
->Enabled = true;
oClient->
```

```
Authentication
```

```
->Token->Enabled = true;
oClient->
```

```
Authentication
```

```
->Token->AuthToken = "your token";
oClient->
```

```
Authentication
```

```
->Basic->Enabled = false;
oClient->
```

```
Authentication
```

```
->URL->Enabled = false;
oClient->
```

```
Authentication
```

```
->Session->Enabled = false;
oClient->Active = true;
```

Authentication Session

First the client connects to the server using an HTTP connection requesting a new Session. If successful, the server returns a SessionId and the client sends this SessionId in the GET HTTP Header of the WebSocket HandShake. Requires setting the UserName and Password and enabling Session Authentication.

```
oClient = new TsgcWebSocketClient();
oClient->
```

```
Authentication
```

```
->Enabled = true;
oClient->
```

```
Authentication
```

```
->Session->Enabled = true;
oClient->
```

```
Authentication
```

```
->User = "your user";
oClient->
```

```
Authentication
```

```
->Password = "your password";
oClient->
```

```
Authentication
```

```
->Basic->Enabled = false;
oClient->
```

```
Authentication
```

```
->URL->Enabled = false;
oClient->
```

```
Authentication
```

```
->Token->Enabled = false;
oClient->Active = true;
```

Authentication URL

This authentication method passes the username and password in the GET HTTP Header of the WebSocket HandShake.

```
oClient = new TsgcWebSocketClient();
oClient->
```

Authentication

```
->Enabled = true;
oClient->
```

Authentication

```
->URL->Enabled = true;
oClient->
```

Authentication

```
->User = "your user";
oClient->
```

Authentication

```
->Password = "your password";
oClient->
```

Authentication

```
->Basic->Enabled = false;
oClient->
```

Authentication

```
->Session->Enabled = false;
oClient->
```

Authentication

```
->Token->Enabled = false;
oClient->Active = true;
```

TsgcWebSocketClient | Client Exceptions

Sometimes there are errors in communications: the server can disconnect a connection because it is not authorized or a message does not have the correct format. There are 2 events where errors are captured.

OnError

This event is fired every time there is an error in WebSocket protocol, like invalid message type, invalid utf8 string... OnError is also fired with **Connection = nil** when WriteData, WriteAndWaitData or Ping cannot send a message because the client is not connected, or, for WriteData and WriteAndWaitData, because the WebSocket handshake has not completed yet. The Error parameter is then `Data not sent, the client is not connected or its handshake has not completed.` No exception is raised to the caller and the client is not disconnected, so always check Connection for nil before using it in this event.

```
void OnError(TsgcWSConnection *Connection, const string Error)
{
  WriteLn("#error: " + Error);
}
```

OnException

This event is fired every time there is an exception, such as writing to a socket that is not active or accessing an object that does not exist.

```
void OnException(TsgcWSConnection *Connection, Exception *E)
{
  WriteLn("#exception: " + E->Message);
}
```

By default, when a **connection is closed by the server**, an **exception will be fired**. If you do not want these exceptions to be fired, disable **Options.RaiseDisconnectExceptions**.
Options.RaiseDisconnectExceptions also controls the OnError report of messages that could not be sent because the client is not connected. Set it to False to drop those messages silently, as earlier versions did.

TsgcWebSocketClient | WebSocket Hand-Shake

The WebSocket protocol uses an HTTP HandShake to upgrade from the HTTP protocol to the WebSocket protocol. This handshake is handled internally by the TsgcWebSocket Client component, but you can add your own custom HTTP headers if the server requires additional HTTP header information.

Example: if you need to add this HTTP Header "Client: sgcWebSockets"

```
void OnHandshake(TsgcWSConnection *Connection, ref TStringList *Headers)
{
  Headers->Add("Client: sgcWebSockets");
}
```

TsgcWebSocketClient | Client Register Protocol

By default, TsgcWebSocketClient does not use any SubProtocol. WebSocket sub-protocols are built on top of the WebSocket protocol and define a custom message protocol. Examples of WebSocket sub-protocols include MQTT, STOMP, etc.

The WebSocket SubProtocol name is sent as an HTTP Header in the WebSocket HandShake. This header is processed by the server, and if the server supports this subprotocol, it will accept the connection. If it is not supported, the connection will be closed automatically.

Example: connect to a websocket server with SubProtocol name 'myprotocol'

```
Client = new TsgcWebSocketClient();
Client->Host = "server host";
Client->Port = server.port;
Client->RegisterProtocol("myprotocol");
Client->Active = true;
```

TsgcWebSocketClient | Client Proxies

TsgcWebSocket client supports connections through proxies. To configure a proxy connection, just fill in the **Proxy** properties of the TsgcWebSocket client.

```
Client = new TsgcWebSocketClient();
Client->Proxy->Enabled = true;
Client->Proxy->Username = "user";
Client->Proxy->Password = "secret";
Client->Proxy->Host = "80.55.44.12";
Client->Proxy->Port = 8080;
Client->Active = true;
```

TsgcTCPClient

TsgcTCPClient is a TCP client component that allows connecting to plain TCP servers.

Introduction

TsgcTCPClient is a TCP client component that allows connecting to plain TCP servers. It provides support for TLS/SSL connections, proxy servers, heartbeat monitoring, automatic reconnection (WatchDog), and TCP keep-alive.

1. Drop a **TsgcTCPClient** component onto the form.

2. Set **Host** and **Port** (default is 80) to connect to an available TCP Server.

3. Set **TLS** to True if you require a secure connection (default is False).

4. Optionally configure **Proxy** if connecting through a proxy server.

5. Handle events: **OnConnect**, **OnDisconnect**, **OnMessage**, **OnBinary**, **OnError**, **OnException**.

6. Set **Active** to True to open the connection.

Reference

- Properties — 23 properties available on the component.
- Methods — 7 public methods.
- Events — 12 events you can handle.

TsgcTCPClient

TsgcTCPClient is a TCP client component that allows connecting to plain TCP servers.

Introduction

TsgcTCPClient is a TCP client component that allows connecting to plain TCP servers. It provides support for TLS/SSL connections, proxy servers, heartbeat monitoring, automatic reconnection (WatchDog), and TCP keep-alive.

1. Drop a **TsgcTCPClient** component onto the form.

2. Set **Host** and **Port** (default is 80) to connect to an available TCP Server.

3. Set **TLS** to True if you require a secure connection (default is False).

4. Optionally configure **Proxy** if connecting through a proxy server.

5. Handle events: **OnConnect**, **OnDisconnect**, **OnMessage**, **OnBinary**, **OnError**, **OnException**.

6. Set **Active** to True to open the connection.

Reference

- Properties — 23 properties available on the component.
- Methods — 7 public methods.
- Events — 12 events you can handle.

TsgcWebSocketServer

TsgcWebSocketServer implements Server WebSocket Component and can handle multiple threaded client connections.

Introduction

TsgcWebSocketServer implements Server WebSocket Component and can handle multiple threaded client connections. Follow the steps below to configure this component:

1. Drop a TsgcWebSocketServer component onto the form

2. Set Port (default is 80). If you are behind a firewall probably you will need to configure it.

3. Set Specifications allowed, by default, all specifications are allowed.

RFC6455: is standard and recommended WebSocket specification.

Hixie76: it's a draft and it's only recommended to establish Hixie76 connections if you want to provide support to old browsers like Safari 4.2

```
void __fastcall TForm1::OnTCPConnectEvent(TsgcWSConnection *aConnection, bool &Accept)
{
    aConnection->Transport = trpTCP;
    Accept = true;
}
```

5. Create a procedure and set property Active = True.

Reference

- Properties — 27 properties available on the component.
- Methods — 7 public methods.
- Events — 22 events you can handle.

Guides

- **Start**
  - Server Start
  - Server Bindings
  - Server Startup - Shutdown
  - Server Keep Active
- **Connections**
  - Server Keep Connections Alive
  - Server Plain TCP
  - Server Close Connection
  - Client Connections
- **Authentication**

- Server Authentication

- **Send Messages**
  - Server Send Text Message
  - Server Send Binary Message

- **Receive Messages**
  - Server Receive Text Message
  - Server Receive Binary Message

- **SSL**
  - Server SSL
  - Server SSL SChannel

TsgcWebSocketServer | Start Server

The first thing you must set when you want to start a server is the listening port. By default, this is set to port 80 but you can change it to any port.
Once the port is set, there are 2 methods to start a server.

Active Property

If you set the Active property to true, the server will start listening for all incoming connections on the configured port.

```
oServer = new TsgcWebSocketServer();
oServer->Port = 80;
oServer->Active = true;
```

If you set the Active property to false, the server will stop and close all active connections.

```
oServer->Active = false;
```

Start / Stop methods

While setting the Active property starts/stops the server in the same thread, the Start and Stop methods are executed in a secondary thread.

```
oServer = new TsgcWebSocketServer();
oServer->Port = 80;
oServer->Start();
```

If you call the Stop() method, the server will stop and close all active connections.

```
oServer->Stop();
```

You can use the method **ReStart**, to Stop and Start server in a secondary thread.

If you change the Port after closing a server, to start listening on a different port, call the method **Bindings.Clear()** after closing the server to delete all previous bindings. Otherwise the server will try to bind to the previous bindings.

TsgcWebSocketServer | Server Bindings

By default, if you only fill **Port property**, the server **binds the listening port on ALL IPs**, so if for example you have 3 IPs: 127.0.0.1, 80.54.11.22 and 12.55.41.17, your server will bind this port on all 3 IPs.
It is usually recommended to bind only to the needed IPs. This is where you can use the Bindings property.
Instead of using the Port property, just use the Bindings property and fill in the required IP and Port.

Example: bind Port 5555 to IP 127.0.0.1 and IP 80.58.25.40

```
oServer = new TsgcWebSocketServer();
binding = oServer->Bindings->Add();
binding->IP = "127.0.0.1";
binding->Port = 5555;
binding = oServer->Bindings->Add();
binding->IP = "80.58.25.40";
binding->Port = 5555;
oServer->Active = true;
```

If you change the Port after closing a server, to start listening on a different port, call the method **Bindings.Clear()** after closing the server to delete all previous bindings. Otherwise the server will try to bind to the previous bindings.

TsgcWebSocketServer | Server Startup Shutdown

Once you have set all required configurations of your server, there are 2 useful events to know when the server has started and when it has stopped.

OnStartup

This event is fired when the server has started and can process new connections.

```
void OnStartup(TObject *Sender)
{
  WriteLn("#server started");
}
```

OnShutdown

This event is fired after the server has stopped and no more connections are accepted.

```
void OnShutdown(TObject *Sender)
{
  WriteLn("#server stopped");
}
```

TsgcWebSocketServer | Server Keep Active

Once the server is started, sometimes it can stop for any reason. If you want to restart the server after an unexpected shutdown, you can use the WatchDog property.

WatchDog

If WatchDog is enabled, when the server detects a shutdown, WatchDog tries to restart every X seconds until the server is active again.

Example: restart every 10 seconds after an unexpected stop with unlimited attempts.

```
oServer = new TsgcWebSocketServer();
oServer->WatchDog->Interval = 10;
oServer->WatchDog->Attempts = 0;
oServer->WatchDog->Enabled = true;
oServer->Active = true;
```

TsgcWebSocketServer | Server SSL

The server can be configured to use **SSL Certificates**. In order to get a production server with a server certificate, you must **purchase** a certificate from a **well-known provider**: Namecheap, GoDaddy, Thawte, etc. For **testing purposes** you can use a **self-signed certificate** (check the Demos/Chat example which uses a self-signed certificate).

Certificate must be in **PEM format**, PEM (from Privacy Enhanced Mail) is defined in RFCs 1421 through 1424, this is a container format that may include just the public certificate (such as with Apache installs, and CA certificate files /etc/ssl/certs), or may include an entire certificate chain including public key, private key, and root certificates. To create a single pem certificate, just open your private key file, copy the contents and paste on certificate file.

Example:

certificate.crt

```
-----BEGIN CERTIFICATE-----
.....
-----END CERTIFICATE-----
```

certificate.key

```
-----BEGIN PRIVATE KEY-----
.....
-----END PRIVATE KEY-----
```

certificate.pem

```
-----BEGIN PRIVATE KEY-----
.....
-----END PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
.....
-----END CERTIFICATE-----
```

To enable SSL, just **enable SSL property** and configure the paths to **CertFile**, **KeyFile** and **RootFile**. If the certificate contains the entire certificate chain (public key, private key, etc.), just set all paths to the same certificate.

Another property you must set is **SSLOptions.Port**, this is the port used for secure connections.

Simple SSL Configuration

Example: configure SSL in IP 127.0.0.1 and Port 443

```
oServer = new TsgcWebSocketServer();
oServer->SSL = true;
oServer->SSLOptions->CertFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->KeyFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->RootCertFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->Active = true;
```

SSL and Non-SSL

You can configure the server to listen on more than one IP and port; check the Binding article which explains how it works. The server can be configured to allow SSL connections and non-SSL connections at the same time (of course, listening on different ports). You only need to bind to two different ports and configure one port for SSL connections and another port for non-SSL connections.

Example: configure server in IP 127.0.0.1, port 80 (non-encrypted) and 443 (SSL)

```
oServer = new TsgcWebSocketServer();
bind = oServer->Bindings->Add();
bind->IP = "127.0.0.1";
bind->Port = 80;
bind = oServer->Bindings->Add();
bind->IP = "127.0.0.1";
bind->Port = 443;
oServer->Port = 80;
oServer->SSL = true;
oServer->SSLOptions->CertFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->KeyFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->RootCertFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->Port = 443;
oServer->Active = true;
```

TsgcWebSocketServer | TLS and plain HTTP on the same port

A TLS server (**TsgcWebSocketServer**, **TsgcWebSocketHTTPServer** and **TsgcHTTPServer** with the default Indy, IOCP or EPOLL IOHandler) can also accept **plain HTTP and WebSocket connections on the same port** it uses for TLS. When this option is enabled, the server peeks at the first byte of every new connection. A `$16` byte (TLS handshake record) means the client starts a TLS handshake, any other byte means the connection is served as plain text. If no byte arrives within `PlainTextTimeout`, the connection is treated as TLS, which is the previous behaviour. With the IOCP and EPOLL IOHandlers the decision is made on the first data received, without blocking, and `PlainTextTimeout` is not used.
The typical use is migrating an existing published plain HTTP port to HTTPS without changing the port configured on the clients. Enable `AllowPlainText` while the clients are updated, and switch it off again once all of them use TLS.

Properties

- **SSLOptions.AllowPlainText**: `Boolean`, default `False`. When `True`, the TLS listener also accepts plain connections on the same port.
- **SSLOptions.PlainTextPolicy**: what the server does with a plain connection.
  - `ptpServe` (default): the plain request is served normally.
  - `ptpRedirect`: the server answers `308 Permanent Redirect` to `https://<Host><path>`, so the client keeps the method and the body. If the request has no Host header, the server answers `400 Bad Request`.
  - `ptpReject`: the server answers `400 Bad Request` and closes the connection.
- **SSLOptions.PlainTextTimeout**: `Integer`, milliseconds, default `5000`. Time to wait for the first byte of the client before the connection is treated as TLS. It only applies to the default Indy IOHandler.
- **Connection.TLS**: read-only `Boolean` of the server connection (`TsgcWSConnectionServer`), `True` when the connection uses TLS. The server method `IsTLS(AContext)` returns the same value from the `TIdContext` of a request.

The enumeration `TsgcSSLPlainTextPolicy` is declared in the unit `sgcWebSocket_Types`.

Example

Start a server on port 443 that accepts TLS and plain connections and serves both.

```
oServer = new TsgcWebSocketHTTPServer(this);
oServer->SSL = true;
oServer->Port = 443;
oServer->SSLOptions->Port = 443;
oServer->SSLOptions->CertFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->KeyFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->RootCertFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->AllowPlainText = true;
oServer->SSLOptions->PlainTextPolicy = ptpServe;
oServer->Active = true;
```

To know whether a request uses TLS, call the server method `IsTLS(AContext)`, for example in the **OnCommandGet** or **OnCommandOther** events. The `TLS` property of the connection (`TsgcWSConnectionServer`) is also available.

```
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo *AResponseInfo)
{
  if (TsgcWebSocketHTTPServer1->IsTLS(AContext))
    AResponseInfo->ContentText = "Secure connection";
  else
    AResponseInfo->ContentText = "Plain connection, please use https://";
}
```

Limitations

- With the **IOCP** IOHandler, **OnConnect** fires before the first byte arrives, so the `TLS` value read inside **OnConnect** is not yet reliable. From the first request on, it is correct.
- The option is ignored by the WebBroker bridge server.
- It is not applicable to the http.sys server (**TsgcWebSocketServer_HTTPAPI**).
- **HSTS** headers (see `SecurityOptions`) are only sent on TLS connections, never on plain ones.

Server SSL | SChannel for Indy Servers

Indy-based server components (**TsgcWebSocketServer**, **TsgcWebSocketHTTPServer**) can use **Windows SChannel** (Secure Channel) as the TLS provider instead of OpenSSL. SChannel is the native Windows TLS implementation, so it does not require any external DLLs.

How it works

When a server component has SSL enabled and the IOHandler is set to **iohSChannel**, the server creates a **TsgcIdServerIOHandlerSSLSChannel** instance that handles all TLS operations using the Windows SChannel API. For every incoming client connection the server performs the TLS handshake through the SChannel provider, negotiating the protocol version, cipher suite, and binding the configured certificate.
SChannel reads certificates from the **Windows Certificate Store** or from a **PFX file** (.pfx / .p12). No PEM files are needed and no OpenSSL libraries need to be deployed.

Configuration

To enable SChannel on an Indy server, configure the following properties:
1. Set the **SSL** property to **True**.
2. Set **SSLOptions.IOHandler** to **iohSChannel**.
3. Set **SSLOptions.Version** to the desired TLS version (tls1_2, tls1_3, ...).
4. Set **SSLOptions.Port** to the port used for secure connections.
5. Configure the certificate using one of the two methods described below.

SChannel_Options properties

The **SSLOptions.SChannel_Options** sub-property exposes the SChannel-specific settings:

Property	Description
CertHash	The thumbprint (hexadecimal hash) of a certificate installed in the Windows Certificate Store.
CertStoreName	Which certificate store to search: **scsnMY** (Personal), scsnRoot, scsnTrust, scsnCA.
CertStorePath	Store location: **scspStoreLocalMachine** or scspStoreCurrentUser.
CipherList	Optional colon-separated list of cipher algorithms (e.g. CALG_AES_256:CALG_AES_128). Empty means use system defaults.
UseLegacyCredentials	When True, uses the legacy SCHANNEL_CRED structure instead of SCH_CREDENTIALS. Enable this for older Windows versions that do not support the newer API.
Revocation.Check	How much of the client certificate chain has its revocation status looked up: **scrcDisabled** (default), scrcEndCertificate, scrcChainExcludeRoot or scrcChain.
Revocation.Timeout	Milliseconds allowed for the whole CRL and OCSP retrieval, not per URL. Default 5000. Zero leaves the Windows default, which can be much longer.
Revocation.CacheOnly	When True, only revocation data Windows has already cached is used and the handshake never goes to the network. Default False.

Revocation.IgnoreRevocationOffline	Accept the client certificate when the responder could not be reached, the soft-fail behaviour. Default True. False rejects instead.
Revocation.IgnoreNoRevocationCheck	Accept a client certificate that publishes no CRL distribution point and no OCSP responder. Default True.

Additionally, the general **SSLOptions** properties **CertFile** and **Password** are used when loading a certificate from a PFX file.

Certificate from Windows Store

If the certificate is already installed in the Windows Certificate Store, provide the certificate **thumbprint** and indicate where it is located.
To find the thumbprint, open **PowerShell** and run:

```
dir cert:\localmachine\my
```

The **Thumbprint** column shows the hexadecimal hash you need.

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint                                Subject
----------                                -------
C12A8FC8AE668F866B48F23E753C93D357E9BE10  CN=*.mydomain.com
```

```
oServer = new TsgcWebSocketServer();
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->Version = tls1_2;
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->SSLOptions->SChannel_Options->CertHash = "C12A8FC8AE668F866B48F23E753C93D357E9BE10";
oServer->SSLOptions->SChannel_Options->CertStoreName = scsnMY;
oServer->SSLOptions->SChannel_Options->CertStorePath = scspStoreLocalMachine;
oServer->Active = true;
```

Certificate from PFX file

If you have a PFX (.pfx or .p12) certificate file, set the **CertFile** and **Password** properties on SSLOptions. SChannel will import the certificate at startup.

```
oServer = new TsgcWebSocketServer();
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->Version = tls1_2;
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->SSLOptions->CertFile = "c:\\certificates\\server.pfx";
oServer->SSLOptions->Password = "mypassword";
oServer->Active = true;
```

If you have a PEM certificate and private key, convert them to PFX format first using OpenSSL:

```
openssl pkcs12 -inkey server.key -in server.crt -export -out server.pfx
```

TLS Version

Use the **SSLOptions.Version** property to control which TLS version the server accepts:

Value	Description
tls1_0	TLS 1.0 (not recommended)
tls1_1	TLS 1.1

tls1_2	TLS 1.2 (recommended)
tls1_3	TLS 1.3
tlsUndefined	Accept TLS 1.0, 1.1 and 1.2

Cipher List

By default SChannel uses the system cipher configuration. You can restrict the allowed ciphers by setting **SChannel_Options.CipherList** to a colon-separated list of algorithm names, for example:

```
CALG_AES_256:CALG_AES_128
```

Leave this property empty to use the Windows defaults.

Legacy Credentials

Windows Server 2019 and earlier may not support the newer **SCH_CREDENTIALS** API. If the server fails to start on an older Windows version, set **SChannel_Options.UseLegacyCredentials** to **True** to use the legacy **SCHANNEL_CRED** structure instead.
The component detects the Windows version automatically in most cases, but you can force legacy mode if needed.

Client Certificate Revocation

When the server validates client certificates (**SSLOptions.VerifyCertificate** set to **True**), the **SChannel_Options.Revocation** sub-properties decide whether the certificate a client presents is also checked against its CRL distribution point or OCSP responder. Revocation is disabled by default. Read Server SSL | SChannel Client Certificate Revocation for the full description of each setting.

Advantages over OpenSSL

- **No external DLLs:** SChannel is built into Windows, so you do not need to deploy libeay32.dll / ssleay32.dll or libcrypto / libssl.
- **Windows Certificate Store integration:** use certificates already installed and managed by the operating system.
- **Automatic updates:** TLS improvements and security patches are applied through Windows Update.

Notes

- SChannel is available on **Windows only**. For cross-platform servers, use OpenSSL (iohOpenSSL).
- The server must have the **private key** associated with the certificate. When using the Windows Store method, the certificate must have been imported with its private key.
- For production servers using the Certificate Store method, the **Local Machine** store (scspStoreLocalMachine) is recommended so the certificate is available regardless of which user account runs the service.
- Only **PFX** (.pfx / .p12) certificate files are supported. If you have PEM files, convert them to PFX format first.

Server SSL | SChannel Client Certificate Revocation

Setting **SSLOptions.VerifyCertificate** to True makes the server ask each incoming client for a certificate and validate the chain it presents. That validation establishes that the client chain is trusted. It says nothing about whether the client certificate has been revoked since it was issued.
That gap matters wherever the certificate is the credential: a laptop is lost or an employee leaves, the certificate issued to them is revoked, and until the day it expires it still opens a session on your server. Revocation is looked up only when it is asked for, through **SSLOptions.SChannel_Options.Revocation**.
The revocation settings are read while a client certificate is being validated, so they do nothing unless **SSLOptions.VerifyCertificate** is True, which is what makes the server request a client certificate in the first place. Whether a client that presents no certificate at all is refused or let through is a separate decision, taken by **SSLOptions.VerifyCertificate_Options.FailIfNoCertificate**.

```
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->VerifyCertificate = true;
oServer->SSLOptions->VerifyCertificate_Options->FailIfNoCertificate = true;
oServer->SSLOptions->SChannel_Options->Revocation->Check = scrcChainExcludeRoot;
```

The status is retrieved while the certificate chain is built, so the CRL distribution point or OCSP responder named by the client certificate is contacted during the handshake. That is why the check is off by default: it puts an outbound request in the path of every connection the server accepts.

Check

Selects how much of the client chain has its revocation status looked up.

scrcDisabled	Default. No revocation status is retrieved and none is required.
scrcEndCertificate	Only the client certificate itself is checked. The cheapest option, one lookup per connection.
scrcChainExcludeRoot	The client certificate and the intermediates that issued it are checked, the root is not. The usual choice, because the root of the CA that issues your client certificates rarely publishes a CRL for itself.
scrcChain	The whole chain including the root.

Timeout

Milliseconds allowed for the whole revocation retrieval, not per URL, so a responder that does not answer cannot stall the handshake while the server waits on it. Default 5000. A value of zero leaves the Windows default in force, which can be considerably longer.

CacheOnly

When True, only revocation data Windows has already cached is used and the handshake never goes to the network, so an incoming connection is never delayed by a responder. A cold cache then leaves the status unknown, which **IgnoreNoRevocationCheck** decides. Default False.

IgnoreRevocationOffline

Decides what happens when the responder could not be reached. Default True, accept the client certificate, which is the soft-fail behaviour. Set it to False to reject instead. Rejecting is stricter, but it makes every new connection depend on the responder being reachable from the server, so a responder outage becomes an outage for all of your clients.

IgnoreNoRevocationCheck

Decides what happens when the client certificate publishes no CRL distribution point and no OCSP responder at all, so its status cannot be established. Default True, accept it. Certificates issued by an internal CA frequently carry no revocation information, so set this to False only once you know that every certificate you hand to a client publishes one.
Neither Ignore property ever accepts a certificate that came back **confirmed revoked**. That outcome is always rejected and the handshake fails.

Reported statuses

When the chain check refuses a client, the reason is one of:

ecCryptRevoked	The client certificate has been revoked by its issuer. Always rejected.
ecCryptRevocationOffline	The responder could not be reached. Only rejected when IgnoreRevocationOffline is False.
ecCryptNoRevocationCheck	The client certificate publishes no revocation information. Only rejected when IgnoreNoRevocationCheck is False.

Secure defaults

Setting **SSLOptions.Preset** to **tlspSecureDefaults** enables the check as **scrcChainExcludeRoot** on the server, matching what the same preset already applied on the client, and keeps the two soft-fail settings, so a client that used to be admitted is still admitted. A scope you chose yourself before applying the preset is left untouched.

```
oServer->SSLOptions->Preset = tlspSecureDefaults;
```

Notes

- The defaults are unchanged, revocation stays disabled, so an existing server behaves exactly as it did before the upgrade. Nothing is looked up until you set **Check** or apply the secure defaults preset.
- SChannel is available on **Windows only**. These settings apply to **iohSChannel** and are ignored by the other IOHandlers.
- See Server SSL | SChannel for Indy Servers for the rest of the SChannel configuration, and Server Verify Certificate for client certificate validation in general.

TsgcWebSocketServer | Server Verify Certificate

By default, the server does not verify peer certificates. To configure the server to verify client certificates, implement the following steps:

1. Set the property SSLOptions.VerifyCertificate = true

Handle the event OnSSLVerifyPeer and implement the following code to be notified every time a client connects with a certificate.

```
void OnSSLSSLVerifyPeer(System::TObject* Sender, TIdX509* Certificate, bool &Accept)
{
    // ... validate the certificate
    if (Certificate_OK)
        Accept = true;
    else
        Accept = false;
}
```

Note that the event **OnSSLVerifyPeer** is **only called if the client provides a certificate**, if a client doesn't provide a certificate, the event is not fired.

You can configure the **server to only allow SSL connections that use a certificate**. To do this, set the following property:

- SSLOptions.VerifyCertificate_Options.FailIfNoCertificate = true

If the client doesn't provide a certificate, the connection will be closed in the SSL Handshake.

TsgcWebSocketServer | Server Keep Connections Alive

Once a client has connected to the server, sometimes the connection can be closed due to poor signal, connection errors, etc. Use HeartBeat to keep the connection alive.

HeartBeat

The **HeartBeat** property allows you to **send a Ping** every X **seconds** to **maintain the connection alive**. Some clients close TCP connections if there is no data exchanged between peers. HeartBeat solves this problem by sending a ping at a specific interval. Usually this is enough to maintain a connection active, but you can set a Time-Out interval if you want to close the connection when a response from the client is not received after X seconds.

Example: send a ping to all connected clients every 30 seconds

```
oServer = new TsgcWebSocketServer();
oServer->HeartBeat->Interval = 30;
oServer->HeartBeat->Timeout = 0;
oServer->HeartBeat->Enabled = true;
oServer->Active = true;
```

TsgcWebSocketServer | Server Plain TCP

The WebSocket server accepts WebSocket, HTTP, SSE, and other protocols, but can also work with plain TCP connections. Read more about TCP Connections.

There are 2 events that can be used to handle TCP connections.

OnTCPConnect

This event is called after a client connects to the server and before any handshake between client and server. The OnConnect event is only fired after the client sends a message (to allow the server to detect which protocol is being used).

This event allows you to know that a new client is trying to connect to the server, and the server can accept or reject the connection. By default, the server always accepts the connection.

OnUnknownProtocol

This event is called when the server receives the first message from a client but cannot detect whether it is any of the known protocols. In this event, the server can accept or reject the protocol.

OnConnect

This event is fired after a successful and complete connection. If the connection is plain TCP, it is fired after the protocol is accepted in the OnUnknownProtocol event.

TsgcWebSocketServer | Server Close Connection

A single Connection can be closed using Close or Disconnect methods.

Disconnect

TsgcWSConnection has a method called Disconnect(), that allows you to disconnect the connection at the socket level. If you call this method, the socket will be disconnected directly without waiting for any response from the client. You can send a Close Code with this method.

Close

TsgcWSConnection has a method called Close(), which allows you to send a message to the client requesting to close the connection. If the client receives this message, it must close the connection and the server will receive a notification that the connection is closed. You can send a Close Code with this method.

DisconnectAll

Disconnects all active connections. This method is called automatically before the server stops listening, but you can call this method at any time.

TsgcWebSocketServer | Client Connections

To access the active client connections, you can use the Connections property to iterate through the list and access the client connection class. The Connections property accesses a threaded list, so first lock the list and when you are finished, unlock the list.

```
void DoClientIPAddresses() {
    TList *oList;
    TsgcWSConnectionServer *oConnection;
    oList = TsgcWebSocketHTTPServer1->LockList();
    try {
        for (int i = 0; i < oList->Count; ++i) {
            oConnection = dynamic_cast<TsgcWSConnectionServer*>(static_cast<TIdContext*>(oList->Items[i])->Data);
            ShowMessage(oConnection->IP + ":" + IntToStr(oConnection->Port));
        }
    } __finally {
        TsgcWebSocketHTTPServer1->UnlockList();
    }
}
```

TsgcWebSocketServer | Server Authentication

TsgcWebSocket server supports 3 types of Authentications:

- **Basic:** reads an HTTP Header during the WebSocket HandShake with User and Password encoded as Basic Authorization.
- **Session:** first the client requests an HTTP session from the server and if the server returns a session, it is passed in the GET HTTP Header of the WebSocket HandShake. (* own authorization method for sgcWebSockets library).
- **URL:** reads the request authorization using the GET HTTP Header of the WebSocket HandShake. (* own authorization method for sgcWebSockets library).

You can set a list of Authenticated users, using **AuthUsers** property, just set your users with the following format: user=password

OnAuthentication

Every time the server receives an Authentication Request from a client, this event is called to indicate whether the user is authenticated or not.
Use Authenticated parameter to accept or not the connection.

```
void OnAuthentication(TsgcWSConnection *Connection, string aUser, string aPassword,
  ref bool Authenticated)
{
  if ((aUser == "user") && (aPassword == "secret"))
  {
    Authenticated = true;
  }
  else
  {
    Authenticated = false;
  }
}
```

OnUnknownAuthentication

If the authentication type is not supported by default, such as JWT, you can still use this event to accept or reject the connection. Just read the parameters and decide whether to accept the connection.

```
void OnUnknownAuthentication(TsgcWSConnection *Connection, string AuthType, string AuthData,
  ref string aUser, ref string aPassword, ref bool Authenticated)
{
  if (AuthType == "Bearer")
  {
    if (AuthData == "jwt_token")
    {
      Authenticated = true;
    }
    else
    {
      Authenticated = false;
    }
  }
  else
  {
    Authenticated = false;
  }
}
```

TsgcWebSocketServer | Server Send Text Message

Once the client has connected to the server, the server can send text messages. To send a Text Message, call the WriteData() method to send a message to a single client, or use Broadcast to send a message to all clients.

Send a Text Message

Call the **WriteData()** method to send a text message.

```
TsgcWebSocketServer1->WriteData("guid", "My First sgcWebSockets Message!.");
```

If **QueueOptions.Text** has a **different value from qmNone**, instead of being processed on the same thread that is called, it will be processed on a secondary thread. By default this option is disabled.

QueueOptions doesn't work if the property IOHandlerOptions.IOHandlerType = iohIOCP (due to the IOCP architecture, this feature is not supported).

You can also call the WriteData() method from **TsgcWSConnection** too, **example:** send a message to client when connects to server.

```
void OnConnect(TsgcWSConnection *Connection)
{
  Connection->WriteData("Hello From Server");
}
```

Send a message to ALL connected clients

Call the **Broadcast()** method to send a text message to all connected clients.

```
TsgcWebSocketServer1->Broadcast("Hello From Server");
```

TsgcWebSocketServer | Server Send Binary Message

Once the client has connected to the server, the server can send binary messages. To send a Binary Message, call the WriteData() method to send a message to a single client, or use Broadcast to send a message to all clients.

Send a Binary Message

Call the **WriteData()** method to send a binary message.

```
TsgcWebSocketServer1->WriteData("guid", new TMemoryStream());
```

If **QueueOptions.Binary** has a **different value from qmNone**, instead of being processed on the same thread that is called, it will be processed on a secondary thread. By default this option is disabled.

QueueOptions doesn't work if the property IOHandlerOptions.IOHandlerType = iohIOCP (due to the IOCP architecture, this feature is not supported).

You can also call the WriteData() method from **TsgcWSConnection** too, **example:** send a message to client when connects to server.

```
void OnConnect(TsgcWSConnection *Connection)
{
  Connection->WriteData(new TMemoryStream());
}
```

Send a message to ALL connected clients

Call the **Broadcast()** method to send a binary message to all connected clients.

```
TsgcWebSocketServer1->Broadcast(new TMemoryStream());
```

TsgcWebSocketServer | Server Receive Text Message

When the server receives a Text Message, the **OnMessage** event is fired. Read the Text parameter to retrieve the string of the message received.

```
void OnMessage(TsgcWSConnection *Connection, const string Text)
{
  ShowMessage("Message Received from Client: " + Text);
}
```

By default, server uses **neAsynchronous** method to dispatch OnMessage event, this means that **this event is executed on the context of Main Thread**, so it's thread-safe to update any control of a form for example.

If your server receives lots of messages or you need to control the synchronization with other threads, set NotifyEvents property to **neNoSync**, this means that OnMessage event will be e**xecuted on the context of connection thread**, so if you require to update any control of a form or access shared objects, you must implement your own synchronization methods.

TsgcWebSocketServer | Server Receive Binary Message

When the server receives a Binary Message, the **OnBinary** event is fired. Read the Data parameter to retrieve the binary message received.

```
void OnBinary(TsgcWSConnection *Connection, const TMemoryStream *Data)
{
  oBitmap = new TBitmap();
  oBitmap->LoadFromStream(Data);
  Image1->Picture->Assign(oBitmap);
  Log(
      "#image uncompressed size: " + IntToStr(Data->Size) +
      ". Total received: " + IntToStr(Connection->RecBytes));
  oBitmap.Free();
}
```

By default, server uses **neAsynchronous** method to dispatch OnMessage event, this means that **this event is executed on the context of Main Thread**, so it's thread-safe to update any control of a form for example.

If your server receives lots of messages or you need to control the synchronization with other threads, set NotifyEvents property to **neNoSync**, this means that OnMessage event will be e**xecuted on the context of connection thread**, so if you require to update any control of a form or access shared objects, you must implement your own synchronization methods.

TsgcWebSocketServer | Server Read Headers from Client

When a **client connects** to the WebSocket server, it sends a list of **headers** with information about the client connection. To read these client headers, you can use the **OnHandshake** event of the server component, which is called when the server receives the headers from the client and before it sends a response.
Client headers are stored in **HeadersRequest** property of **TsgcWSConnectionServer**.

```
void OnServerHandshake(TsgcWSConnection *Connection, TStringList *Headers)
{
  ShowMessage(dynamic_cast<TsgcWSConnectionServer*>(Connection)->HeadersRequest->Text);
}
```

TsgcWebSocketHTTPServer

AResponseInfo.

Introduction

TsgcWebSocketHTTPServer implements Server WebSocket Component and can handle multiple threaded client connections as TsgcWebSocketServer, and allows you to serve HTML pages using a built-in HTTP Server, sharing the same port for WebSocket connections and HTTP requests.

Follow the steps below to configure this component:

1. Drop a TsgcWebSocketHTTPServer component in the form

2. Set Port (default is 80). If you are behind a firewall probably you will need to configure it.

3. Set Specifications allowed, by default, all specifications are allowed.

RFC6455: is standard and recommended WebSocket specification.

Hixie76: it's a draft and it's only recommended to establish Hixie76 connections if you want to provide support to old browsers like Safari 4.2

```
AResponseInfo.ContentText := '<HTML><HEADER>TEST</HEAD><BODY>Hello!</BODY></
HTML>';
```

OnHTTPUploadBeforeCreatePostStream: this event is called after the headers have been read and before the post stream is created.

* In some cases, you may get a high consume of cpu due to unsolicited connections, in these cases, just return an error 500 if it's a HTTP request or close connection for Unknown Protocol requests.

5. Create a procedure and set property Active = true.

Reference

- Properties — 36 properties available on the component.
- Methods — 9 public methods.
- Events — 36 events you can handle.

Guides

- **HTTP**
  - HTTP Server Requests
  - HTTP Dispatch Files
  - HTTP/2 Server
  - HTTP/2 Server Push
  - HTTP/2 Alternate Service
  - HTTP/2 Server Threads
  - HTTP Post Big Files
  - HTTP 404 Error without Response Body
- **SSL**

    - Server SSL SChannel
- **Other**
    - HTTP Server Sessions

TsgcWebSocketHTTPServer | HTTP Server Requests

Use OnCommandGet to handle HTTP client requests. Use the following parameters:

- **RequestInfo:** contains HTTP request information.
- **ResponseInfo:** is the HTTP response to HTTP Request.
  - **ContentText:** is the response in text format.
  - **ContentType:** is the type of Content-Type.
  - **ResponseNo:** number of HTTP response, example: 200.

```
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Document == "/")
  {
    AResponseInfo->ContentText = "<html><head><title>Test Page</title></head><body></body></html>";
    AResponseInfo->ContentType = "text/html";
    AResponseInfo->ResponseNo = 200;
  }
}
```

OnBeforeCommand

Use this event to customize the HTTP response. For example, if you want some endpoints to use an authorization scheme while others can be accessed without authorization, use the options parameter to allow or disable it. Below is an example where Authorization Basic is enabled, but when a user requests the endpoint /public, authorization is not required.

```
void __fastcall TForm1::OnBeforeCommand(TsgcWSConnection* aConnection, TIdHTTPRequestInfo* ARequestInfo, TIdHTTPR
{
    if (ARequestInfo->Document == "/public")
        aOptions << hcoAuthorizedBasic;
}
```

TsgcWebSocketHTTPServer | HTTP Dispatch Files

When a client requests a file, the **OnCommandGet** event is fired, but you can use the **DocumentRoot** property to dispatch files automatically.

Example: if you set **DocumentRoot** to **c:/www/files**. Every time a new file is requested, the server will search in this folder. If the file exists, it will be dispatched automatically.

TsgcWebSocketHTTPServer | HTTP/2 Server

sgcWebSockets HTTP Server allows you to handle HTTP/1.1 and HTTP/2.0 requests, you can enable HTTP/2 protocol using HTTP2Options of Server.

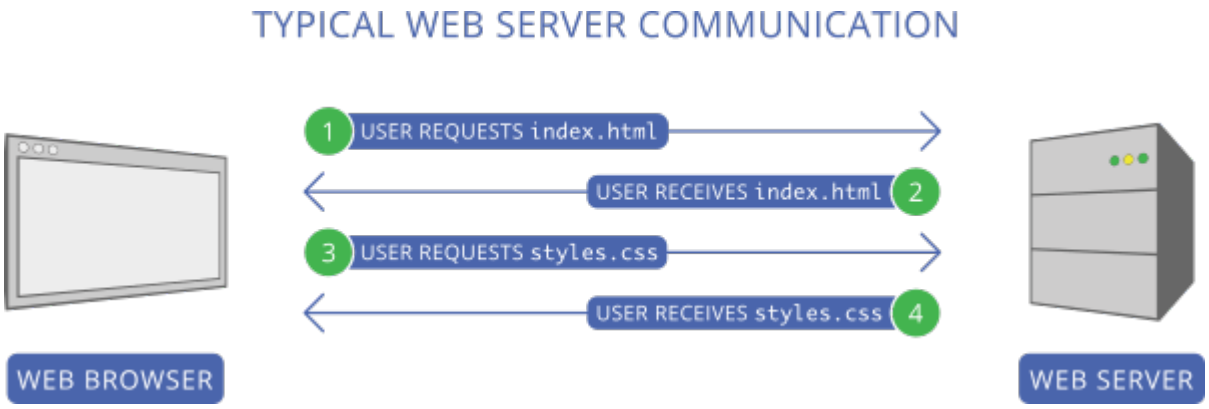
Set **HTTP2Options.Enabled = true** to allow the server to accept HTTP/2 protocol requests. The requests can be processed by the user exactly the same as with the HTTP/1.1 protocol, read more.

When the HTTP/2 protocol is enabled, the server will still support HTTP/1.1 requests.

By default, OnConnect and OnDisconnect events won't be called when there is a new HTTP/2 connection, but this can be modified by accessing the HTTP2Options.Events properties, where you can customize whether you want to be notified every time there is a new HTTP/2 connection and/or disconnection.

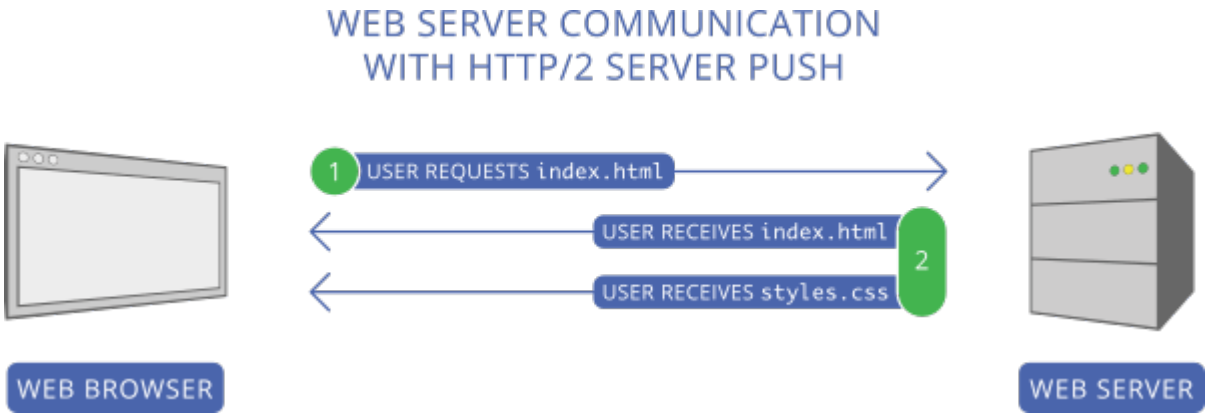
TsgcWebSocketHTTPServer | HTTP/2 Server Push

HTTP usually works with a Request/Response pattern, where the client sends a REQUEST for a resource to the SERVER, and the SERVER sends a RESPONSE with the resource requested or an error. Usually the client, such as a browser, makes a series of requests for assets that are provided by the server.



The main problem with this approach is that the client must first send a request to get the resource (for example, index.html), wait until the server sends the response, read the content, and then make all other requests (for example, styles.css).

HTTP/2 server push tries to solve this problem. When the client requests a file, if the server determines that this file requires additional files, those files will be PUSHED to the client automatically.



In the screenshot above, the client first requests index.html. The server reads this request and sends 2 files as the response: index.html and styles.css, thus avoiding a second request to get styles.css.

Configure Server Push

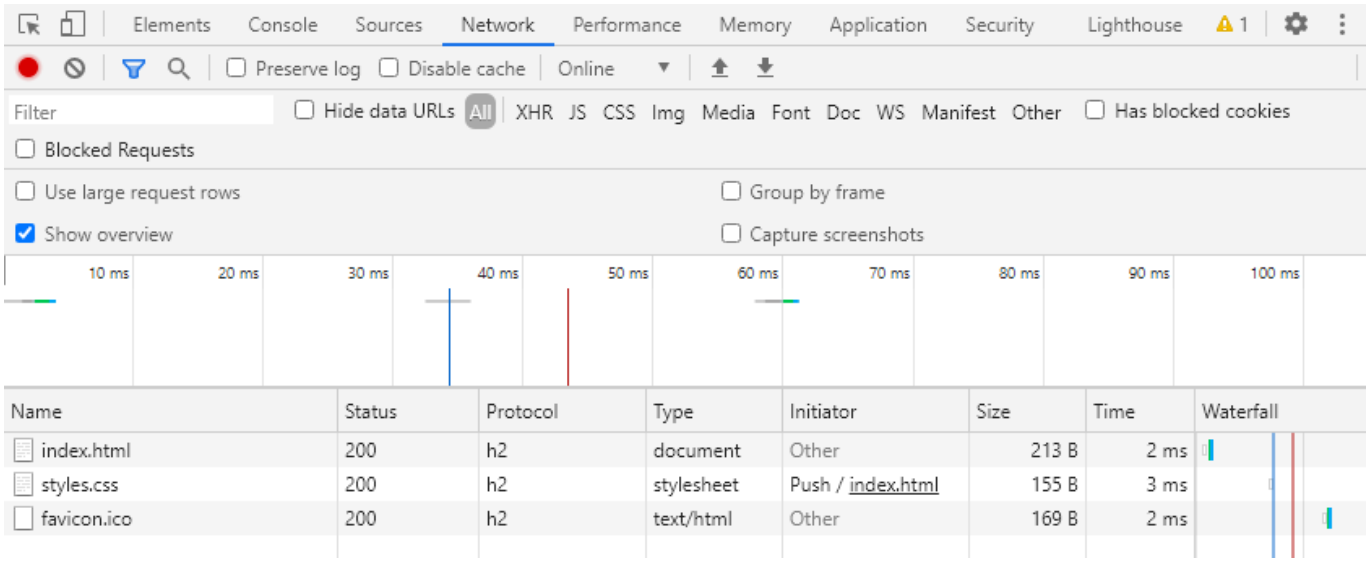
Following the screenshots above, you can configure your server so that every time there is a new request for the / index.html file, the server will send index.html and styles.css.

Use the method **PushPromiseAddPreLoadLinks** to associate each request with a push promise list.

```
TsgcWebSocketHTTPServer *server = new TsgcWebSocketHTTPServer(this);
TStringList *oLinks = new TStringList();
try
{
  oLinks->Add("/styles.css");
  server->PushPromiseAddPreLoadLinks("/index.html", oLinks);
```

```
}
__finally
{
  oLinks->Free();
}

void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Document == "/index.html")
  {
    AResponseInfo->ContentText = "";
    AResponseInfo->ContentType = "text/html";
    AResponseInfo->ResponseNo = 200;
  }
  else if (ARequestInfo->Document == "/styles.css")
  {
    AResponseInfo->ContentText = "";
    AResponseInfo->ContentType = "text/css";
    AResponseInfo->ResponseNo = 200;
  }
}
```

Using the Chrome developer tool, you can view how the styles.css file is pushed to the client.



TsgcWebSocketHTTPServer | HTTP/2 Alternate Service

The **Alt-Svc HTTP header** is used to **inform** the clients that the **same resource can be reached from another service or protocol**, this is useful if you want to inform the HTTP clients that your server supports HTTP/2 for example.

Example: if your server is running on a local IP 127.0.0.1 and is listening on 2 ports: 80 (non encrypted) and 443 (encrypted). You can inform the clients, that HTTP/2 is supported on port 443 using the following HTTP header

```
Alt-Svc: h2=":443"
```

When HTTP/2 is enabled, this header is automatically added if the connection is not running on the HTTP/2 protocol.
You can enable or disable this feature using the property **HTTP2Options.AltSvc**.

TsgcWebSocketHTTPServer | HTTP/2 Server Threads

See below the differences between HTTP 1.1 and HTTP 2.0:

HTTP 1.1

In traditional HTTP behavior, when making multiple requests over the same connection, the client has to wait for the response of each request before sending the next one. This sequential approach significantly increases the load time of a website's resources. To address this issue, HTTP/1.1 introduced a feature called pipelining, allowing a client to send multiple requests without waiting for the server's responses. The server, in turn, responds to the client in the same order as it received the requests.

While pipelining appeared to be a solution, it faced challenges:

- **Server Ignorance or Response Corruption:** Some servers either ignored pipelined requests or corrupted the responses, leading to unreliable communication.
- **Head-of-Line Blocking:** The first request in the pipeline could block subsequent requests, causing a delay in the processing of other requests. This phenomenon, known as head-of-line blocking, resulted in slower page loading times.

In an effort to optimize page loading from servers supporting HTTP/1.1, web browsers implemented a workaround. They open six to eight parallel connections to the server, enabling the simultaneous transmission of multiple requests. This parallelism aims to mitigate the issues associated with pipelining and improve overall page load times.

The choice of six to eight parallel connections by web browsers is based on optimization considerations. The specific reasons behind selecting this number may involve a trade-off between resource utilization, network efficiency, and avoiding potential bottlenecks.

HTTP 2.0

In response to the constraints encountered in pipelining, HTTP/2 introduced a feature called multiplexing. **Multiplexing** allows for **more efficient communication** between the client and server by enabling the **concurrent transmission** of **multiple requests** and responses **over a single connection**.

HTTP/2 utilizes a binary framing mechanism, which means that HTTP messages are broken down into smaller, independent units called frames. These frames can be interleaved and sent over the connection independently of one another. At the receiving end, the frames are reassembled to reconstruct the original HTTP message.

This binary framing mechanism is fundamental to achieving multiplexing in HTTP/2. It enables the browser to send multiple requests over the same connection without encountering blocking issues. As a result, browsers like Chrome utilize the same connection ID for HTTP/2 requests, allowing for efficient and uninterrupted communication between the client and server.
In essence, HTTP/2's multiplexing feature, enabled by the binary framing mechanism, enhances the efficiency and speed of data exchange between clients and servers by facilitating concurrent transmission of multiple requests and responses over a single connection.

TsgcWebSocketHTTPServer

To improve the performance of the HTTP/2 protocol, the requests are dispatched by default in a pool of threads (by default 32) every time a new HTTP/2 request is received by the server. This avoids waits when a single connection

sends many concurrent requests that would require sequential processing (in the context of the connection thread) in the absence of this pool of threads.

The behavior of the pool of threads can be configured with the following properties.

- **HTTP2Options.PoolOfThreads.Enabled:** (by default false) enable this to dispatch HTTP/2 requests in the pool of threads instead of the connection thread.
- **HTTP2Options.Threads:** (by default 32) the number of threads used to handle HTTP/2 requests. Set a number according to the number of processors on your server.

To **fine-tune the requests**, selecting which requests must be processed in the pool of threads (because they are time-consuming) while others can be processed in the connection thread, you can use the event **OnHttp2BeforeAsyncRequest**. This event is raised before queuing the request in the pool of threads. Use the parameter **Async** to set whether the request is threaded or not.

```
void __fastcall TForm1::OnHTTP2BeforeAsyncRequest(TObject *Sender, TsgcWSConnection *Connection, const TIdHTTPRec
{
    if (ARequestInfo.Document == "/fast-request")
        Async = false;
}
```

TsgcWebSocketHTTPServer | 404 Error without Response Body

By default, the Indy library adds a content body to HTTP responses if there is no ContentText or ContentStream assigned. If you want to return an empty response body (for a 404 error or similar), you can use the following approach.

Create a new TStringStream without content and assign it to the ContentStream property of the HTTP Response. This way the HTTP response will be sent without the default HTML tags.

Example

```
private void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  AResponseInfo->ContentStream = new TStringStream("");
  AResponseInfo->ContentType = "text/html";
  AResponseInfo->ResponseNo = 404;
}
```

TsgcWebSocketHTTPServer | Sessions

HTTP is a stateless protocol (at least up to HTTP 1.1), so the client requests a file, the server sends a response, and the connection is closed (you can enable keep-alive so the connection is not closed immediately, but that is beyond the scope of this article). Sessions allow you to store information about the client, which can be used during a client login for example. You can use any unique session ID, search the list of sessions to see if one already exists, and if not, create a new session. A session can be destroyed after a period of inactivity or manually after client logout.

Configuration

There are some properties in TsgcWebSocketHTTPServer that enable/disable sessions in the server component. The most important are:

Property	Description
SessionState	This is the first property that must be enabled in order to use Sessions. Without this prop abled, sessions will not work
SessionTimeout	Here you must set a value greater than zero (in milliseconds) for the maximum time a se be active.
AutoStartSession	Sessions can be created automatically (AutoStartSession = true) or manually (AutoStartSe false). If sessions are created automatically, the server will use RemoteIP as a unique ide check if there is an active session stored.
SessionClass	Optional. The class the server uses when it creates a new session. Set it to your own **TIdH sion** descendant to store your own data inside every session. Must be set before the ser tivated.
SessionList	The list that holds the active sessions. Read it to search, create or remove sessions by co can also assign your own list if you need full control over where sessions are stored. Mus signed before the server is activated.

```
TsgcWebSocketHTTPServer1->SessionState = true;
TsgcWebSocketHTTPServer1->SessionTimeout = 600000;
TsgcWebSocketHTTPServer1->AutoStartSession = false;
```

Create Session

To create a new session, you must create a new **session ID** that is **unique**. You can use any value. **Example:** if the client is authenticating, you can use user + password + remoteip as the session ID.
Then, search the session list to check if it already exists. If it does not exist, create a new one.

When a new session is created **OnSessionStart** event is called and when session is closed, **OnSessionEnd** event is raised.

```
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Document == "/")
  {
    AResponseInfo->ServeFile(AContext, "yourpathhere\\index.html");
  }
```

```
  else
  {
    // check if user is valid
    if (((ARequestInfo->AuthUsername == "user") && (ARequestInfo->AuthPassword == "pass")) == false)
    {
      AResponseInfo->AuthRealm = "Authenticate";
    }
    else
    {
      // create a new session id with authentication data
      String vID = ARequestInfo->AuthUsername + "_" + ARequestInfo->AuthPassword + "_" + ARequestInfo->RemoteIP;

      // search session
      TIdHTTPSession *oSession = TsgcWebSocketHTTPServer1->SessionList->GetSession(vID, ARequestInfo->RemoteIP);

      // create new session if not exists
      if (oSession == NULL)
      {
        oSession = TsgcWebSocketHTTPServer1->SessionList->CreateSession(ARequestInfo->RemoteIP, vID);
      }

      AResponseInfo->ContentText = "<html><head></head><body>Authenticated</body></html>";
      AResponseInfo->ResponseNo = 200;
    }
  }
}
```

Read the Current Session

Once a session exists, the server attaches it to every request that carries the session cookie. Read it from **ARequestInfo.Session**, which is nil when the request has no session.

```
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Session != NULL)
  {
    // Content is a TStrings you can use to store your own values
    int vVisits = StrToIntDef(ARequestInfo->Session->Content->Values["visits"], 0) + 1;
    ARequestInfo->Session->Content->Values["visits"] = IntToStr(vVisits);

    AResponseInfo->ContentText = "Session " + ARequestInfo->Session->SessionID +
      " visits: " + IntToStr(vVisits);
  }
  else
  {
    AResponseInfo->ContentText = "No session";
  }
}
```

Use Your Own Session Class

If you want to keep your own fields inside every session, instead of using the **Content** string list, create a descendant of **TIdHTTPSession** and tell the server to use it through the **SessionClass** property. Set it **before** the server is activated.
The server keeps taking care of everything else: it generates the unique session ID, it sends the session cookie, it applies SessionTimeout and it removes stale sessions. Override the virtual constructor **CreateInitialized** if you want to initialize your own fields when the session is created.

```
class TMySession : public TIdHTTPSession
{
private:
  String FUserName;
  TDateTime FLoginTime;
public:
  __fastcall virtual TMySession(TIdHTTPCustomSessionList *AOwner,
    const String SessionID, const String RemoteIP)
    : TIdHTTPSession(AOwner, SessionID, RemoteIP)
  {
    FLoginTime = Now();
  }
  __property String UserName = {read = FUserName, write = FUserName};
  __property TDateTime LoginTime = {read = FLoginTime, write = FLoginTime};
};

// configure the server before it starts
```

```
TsgcWebSocketHTTPServer1->SessionState = true;
TsgcWebSocketHTTPServer1->SessionTimeout = 600000;
TsgcWebSocketHTTPServer1->SessionClass = __classid(TMySession);
TsgcWebSocketHTTPServer1->Active = true;

// and read it back in any request
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  TMySession *oSession = dynamic_cast<TMySession *>(ARequestInfo->Session);
  if (oSession != NULL)
  {
    AResponseInfo->ContentText = oSession->UserName;
  }
}
```

Use Your Own Session List

If you need full control over how sessions are stored, for example keeping them in a database or sharing them between several servers, assign your own list to the **SessionList** property, before the server is activated. Descend from **TIdHTTPDefaultSessionList** and override the virtual **CreateSession** method. The server calls it internally from CreateUniqueSession, so the unique session ID is still generated for you.
When you assign your own SessionList, the **SessionClass** property no longer applies, because your list decides which class to create.

```
class TMySessionList : public TIdHTTPDefaultSessionList
{
public:
  virtual TIdHTTPSession * __fastcall CreateSession(const String RemoteIP,
    const String SessionID)
  {
    TIdHTTPSession *oSession = new TMySession(this, SessionID, RemoteIP);
    SessionList->Add(oSession);
    return oSession;
  }
};

// assign it before the server starts
TsgcWebSocketHTTPServer1->SessionList = new TMySessionList(NULL);
TsgcWebSocketHTTPServer1->Active = true;
```

A Note About OnCreateSession

The OnCreateSession event lets you return a session instance yourself, but it does **not** assign a session ID to it. If you create the session with a plain constructor, the session ID and the session cookie are empty and the session can never be found again on the next request. Use **SessionClass** instead, which is simpler and handles all of that for you.
Also, do not call **SessionList.CreateUniqueSession** inside OnCreateSession. That method already adds the new session to the list, and the server adds it again when your handler returns, so the same session ends up twice in the list.

TsgcWebSocketHTTPServer | Stream Video

If you want to stream a video file using the server, you can use the function IndyStreamFileVideo to stream the file using chunked transfer encoding.

The function takes the following parameters:

- **AContext:** it's taken from OnCommandGet event.
- **AResponseInfo:** it's taken from OnCommandGet event.
- **aFileName:** it's the full filename of the video to stream.
- **ContentType:** by default is "video/mpeg".
- **aBufferSize:** by default is 1024 bytes.

Example: stream a video when a user goes to the document /video.mp4 and this video is in the folder c:\videos\video.mp4

```
void OnServerCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo *AResponseInf
{
  if (ARequestInfo->Document == "/video.mp4") {
    IndyStreamFileVideo(Acontext, AResponseInfo, "C:\\videos\\video.mp4");
  } else {
    AResponseInfo->ResponseNo = 404;
  }
}
```

Server SSL | SChannel for Indy Servers

Indy-based server components (**TsgcWebSocketServer**, **TsgcWebSocketHTTPServer**) can use **Windows SChannel** (Secure Channel) as the TLS provider instead of OpenSSL. SChannel is the native Windows TLS implementation, so it does not require any external DLLs.

How it works

When a server component has SSL enabled and the IOHandler is set to **iohSChannel**, the server creates a **TsgcIdServerIOHandlerSSLSChannel** instance that handles all TLS operations using the Windows SChannel API. For every incoming client connection the server performs the TLS handshake through the SChannel provider, negotiating the protocol version, cipher suite, and binding the configured certificate.
SChannel reads certificates from the **Windows Certificate Store** or from a **PFX file** (.pfx / .p12). No PEM files are needed and no OpenSSL libraries need to be deployed.

Configuration

To enable SChannel on an Indy server, configure the following properties:
1. Set the **SSL** property to **True**.
2. Set **SSLOptions.IOHandler** to **iohSChannel**.
3. Set **SSLOptions.Version** to the desired TLS version (tls1_2, tls1_3, ...).
4. Set **SSLOptions.Port** to the port used for secure connections.
5. Configure the certificate using one of the two methods described below.

SChannel_Options properties

The **SSLOptions.SChannel_Options** sub-property exposes the SChannel-specific settings:

Property	Description
CertHash	The thumbprint (hexadecimal hash) of a certificate installed in the Windows Certificate Store.
CertStoreName	Which certificate store to search: **scsnMY** (Personal), scsnRoot, scsnTrust, scsnCA.
CertStorePath	Store location: **scspStoreLocalMachine** or scspStoreCurrentUser.
CipherList	Optional colon-separated list of cipher algorithms (e.g. CALG_AES_256:CALG_AES_128). Empty means use system defaults.
UseLegacyCredentials	When True, uses the legacy SCHANNEL_CRED structure instead of SCH_CREDENTIALS. Enable this for older Windows versions that do not support the newer API.

Additionally, the general **SSLOptions** properties **CertFile** and **Password** are used when loading a certificate from a PFX file.

Certificate from Windows Store

If the certificate is already installed in the Windows Certificate Store, provide the certificate **thumbprint** and indicate where it is located.
To find the thumbprint, open **PowerShell** and run:

```
dir cert:\localmachine\my
```

The **Thumbprint** column shows the hexadecimal hash you need.

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint                                Subject
----------                                -------
C12A8FC8AE668F866B48F23E753C93D357E9BE10  CN=*.mydomain.com
```

```
oServer = new TsgcWebSocketHTTPServer();
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->Version = tls1_2;
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->SSLOptions->SChannel_Options->CertHash = "C12A8FC8AE668F866B48F23E753C93D357E9BE10";
oServer->SSLOptions->SChannel_Options->CertStoreName = scsnMY;
oServer->SSLOptions->SChannel_Options->CertStorePath = scspStoreLocalMachine;
oServer->Active = true;
```

Certificate from PFX file

If you have a PFX (.pfx or .p12) certificate file, set the **CertFile** and **Password** properties on SSLOptions. SChannel will import the certificate at startup.

```
oServer = new TsgcWebSocketHTTPServer();
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->Version = tls1_2;
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->SSLOptions->CertFile = "c:\\certificates\\server.pfx";
oServer->SSLOptions->Password = "mypassword";
oServer->Active = true;
```

If you have a PEM certificate and private key, convert them to PFX format first using OpenSSL:

```
openssl pkcs12 -inkey server.key -in server.crt -export -out server.pfx
```

TLS Version

Use the **SSLOptions.Version** property to control which TLS version the server accepts:

Value	Description
tls1_0	TLS 1.0 (not recommended)
tls1_1	TLS 1.1
tls1_2	TLS 1.2 (recommended)
tls1_3	TLS 1.3
tlsUndefined	Accept TLS 1.0, 1.1 and 1.2

Cipher List

By default SChannel uses the system cipher configuration. You can restrict the allowed ciphers by setting **SChannel_Options.CipherList** to a colon-separated list of algorithm names, for example:

```
CALG_AES_256:CALG_AES_128
```

Leave this property empty to use the Windows defaults.

Legacy Credentials

Windows Server 2019 and earlier may not support the newer **SCH_CREDENTIALS** API. If the server fails to start on an older Windows version, set **SChannel_Options.UseLegacyCredentials** to **True** to use the legacy **SCHANNEL_CRED** structure instead.

The component detects the Windows version automatically in most cases, but you can force legacy mode if needed.

Advantages over OpenSSL

- **No external DLLs:** SChannel is built into Windows, so you do not need to deploy libeay32.dll / ssleay32.dll or libcrypto / libssl.
- **Windows Certificate Store integration:** use certificates already installed and managed by the operating system.
- **Automatic updates:** TLS improvements and security patches are applied through Windows Update.

Notes

- SChannel is available on **Windows only**. For cross-platform servers, use OpenSSL (iohOpenSSL).
- The server must have the **private key** associated with the certificate. When using the Windows Store method, the certificate must have been imported with its private key.
- For production servers using the Certificate Store method, the **Local Machine** store (scspStoreLocalMachine) is recommended so the certificate is available regardless of which user account runs the service.
- Only **PFX** (.pfx / .p12) certificate files are supported. If you have PEM files, convert them to PFX format first.

TsgcHTTPServer

Plain HTTP server component — the same engine as TsgcWebSocketHTTPServer, without WebSocket support, for publishing HTTP-only services such as an MCP server.

Introduction

TsgcHTTPServer is a plain HTTP server component: it shares the same underlying engine as TsgcWebSocketHTTPServer (document root, sessions, keep-alive, HTTP compression, HTTP/2, HTTP/3, TLS, authentication, thread pool, IOCP/EPOLL engines, file uploads, firewall, rate limiter...) but it never upgrades a connection to the WebSocket protocol. There is no handshake, no frames, and none of the WebSocket-only members (Extensions, Specifications, HeartBeat, QueueOptions, OnMessage/OnBinary/OnFragmented/OnHandshake, LoadBalancer/OnLoadBalancer*, Groups) are published on it.
Use TsgcHTTPServer when your application only needs to publish an HTTP endpoint and you do not want to link any WebSocket code at all, for example to host an MCP (Model Context Protocol) server, a REST/JSON API, or a plain file/document server. If you also need WebSocket support on the same port, use TsgcWebSocketHTTPServer instead.
Follow the steps below to configure this component:
1. Drop a TsgcHTTPServer component in the form
2. Set Port (default is 80). If you are behind a firewall probably you will need to configure it.
3. Handle OnCommandGet / OnCommandOther to answer requests, or set DocumentRoot to serve static files automatically.

```
AResponseInfo.ContentText := '<HTML><HEADER>TEST</HEAD><BODY>Hello!</BODY></
HTML>';
```

OnHTTPUploadBeforeCreatePostStream: this event is called after the headers have been read and before the post stream is created.

* In some cases, you may get a high consume of cpu due to unsolicited connections, in these cases, just return an error 500 or close the connection from OnUnknownProtocol.

4. Create a procedure and set property Active = true.

Reference

- Properties — 36 properties available on the component.
- Methods — 6 public methods.
- Events — 29 events you can handle.

Guides

- **HTTP**
  - HTTP Server Requests
  - HTTP Dispatch Files
  - HTTP/2 Server
  - HTTP/2 Server Push
  - HTTP/2 Alternate Service
  - HTTP/2 Server Threads
  - HTTP Post Big Files
  - HTTP 404 Error without Response Body
- **SSL**
  - Server SSL SChannel
- **Other**

- HTTP Server Sessions
- HTTP Server Stream Video

TsgcHTTPServer | HTTP Server Requests

Use OnCommandGet to handle HTTP client requests. Use the following parameters:

- **RequestInfo:** contains HTTP request information.
- **ResponseInfo:** is the HTTP response to HTTP Request.
  - **ContentText:** is the response in text format.
  - **ContentType:** is the type of Content-Type.
  - **ResponseNo:** number of HTTP response, example: 200.

```
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Document == "/")
  {
    AResponseInfo->ContentText = "<html><head><title>Test Page</title></head><body></body></html>";
    AResponseInfo->ContentType = "text/html";
    AResponseInfo->ResponseNo = 200;
  }
}
```

OnBeforeCommand

Use this event to customize the HTTP response. For example, if you want some endpoints to use an authorization scheme while others can be accessed without authorization, use the options parameter to allow or disable it. Below is an example where Authorization Basic is enabled, but when a user requests the endpoint /public, authorization is not required.

```
void __fastcall TForm1::OnBeforeCommand(TsgcWSConnection* aConnection, TIdHTTPRequestInfo* ARequestInfo, TIdHTTPR
{
    if (ARequestInfo->Document == "/public")
        aOptions << hcoAuthorizedBasic;
}
```

TsgcHTTPServer | HTTP Dispatch Files

When a client requests a file, the **OnCommandGet** event is fired, but you can use the **DocumentRoot** property to dispatch files automatically.
Example: if you set **DocumentRoot** to **c:/www/files**. Every time a new file is requested, the server will search in this folder. If the file exists, it will be dispatched automatically.

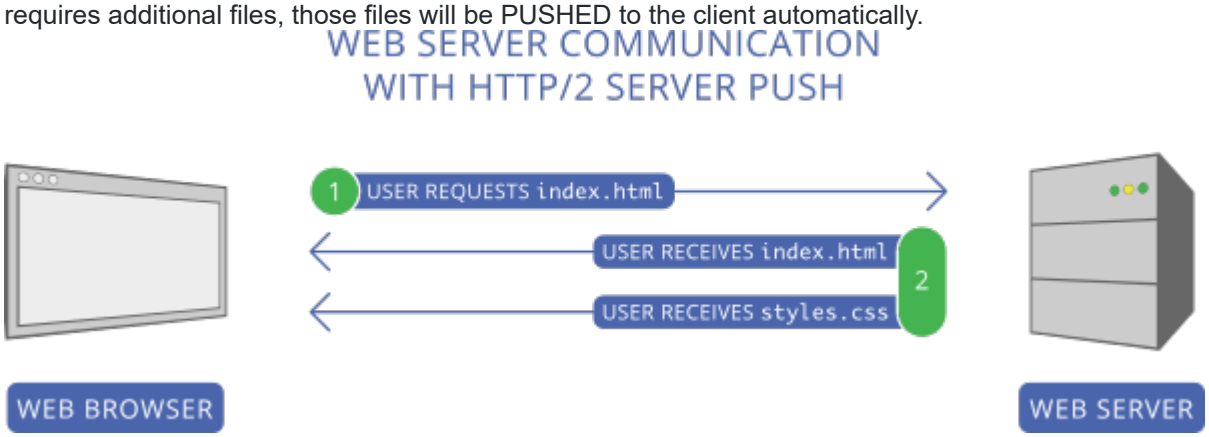
TsgcHTTPServer | HTTP/2 Server

sgcWebSockets HTTP Server allows you to handle HTTP/1.1 and HTTP/2.0 requests, you can enable HTTP/2 protocol using HTTP2Options of Server.
Set **HTTP2Options.Enabled = true** to allow the server to accept HTTP/2 protocol requests. The requests can be processed by the user exactly the same as with the HTTP/1.1 protocol, read more.
When the HTTP/2 protocol is enabled, the server will still support HTTP/1.1 requests.
By default, OnConnect and OnDisconnect events won't be called when there is a new HTTP/2 connection, but this can be modified by accessing the HTTP2Options.Events properties, where you can customize whether you want to be notified every time there is a new HTTP/2 connection and/or disconnection.

TsgcHTTPServer | HTTP/2 Server Push

HTTP usually works with a Request/Response pattern, where the client sends a REQUEST for a resource to the SERVER, and the SERVER sends a RESPONSE with the resource requested or an error. Usually the client, such as a browser, makes a series of requests for assets that are provided by the server.

The main problem with this approach is that the client must first send a request to get the resource (for example, index.html), wait until the server sends the response, read the content, and then make all other requests (for example, styles.css).
HTTP/2 server push tries to solve this problem. When the client requests a file, if the server determines that this file requires additional files, those files will be PUSHED to the client automatically.



In the screenshot above, the client first requests index.html. The server reads this request and sends 2 files as the response: index.html and styles.css, thus avoiding a second request to get styles.css.

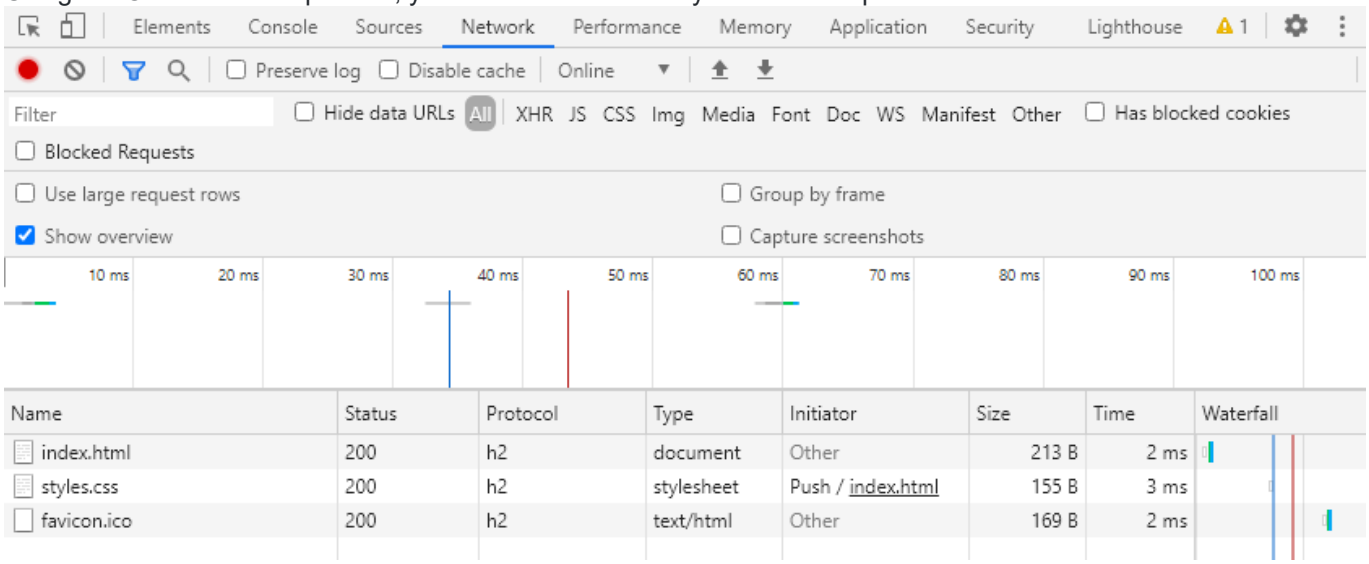
Configure Server Push

Following the screenshots above, you can configure your server so that every time there is a new request for the / index.html file, the server will send index.html and styles.css.
Use the method **PushPromiseAddPreLoadLinks** to associate each request with a push promise list.

```
TsgcHTTPServer *server = new TsgcHTTPServer(this);
TStringList *oLinks = new TStringList();
try
{
  oLinks->Add("/styles.css");
  server->PushPromiseAddPreLoadLinks("/index.html", oLinks);
}
__finally
{
  oLinks->Free();
}

void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Document == "/index.html")
  {
    AResponseInfo->ContentText = "";
    AResponseInfo->ContentType = "text/html";
    AResponseInfo->ResponseNo = 200;
  }
```

```
  else if (ARequestInfo->Document == "/styles.css")
  {
    AResponseInfo->ContentText = "";
    AResponseInfo->ContentType = "text/css";
    AResponseInfo->ResponseNo = 200;
  }
}
```

Using the Chrome developer tool, you can view how the styles.css file is pushed to the client.



TsgcHTTPServer | HTTP/2 Alternate Service

The **Alt-Svc HTTP header** is used to **inform** the clients that the **same resource can be reached from another service or protocol**, this is useful if you want to inform the HTTP clients that your server supports HTTP/2 for example.
Example: if your server is running on a local IP 127.0.0.1 and is listening on 2 ports: 80 (non encrypted) and 443 (encrypted). You can inform the clients, that HTTP/2 is supported on port 443 using the following HTTP header
`Alt-Svc: h2=":443"`
When HTTP/2 is enabled, this header is automatically added if the connection is not running on the HTTP/2 protocol.
You can enable or disable this feature using the property **HTTP2Options.AltSvc**.

TsgcHTTPServer | HTTP/2 Server Threads

See below the differences between HTTP 1.1 and HTTP 2.0:

HTTP 1.1

In traditional HTTP behavior, when making multiple requests over the same connection, the client has to wait for the response of each request before sending the next one. This sequential approach significantly increases the load time of a website's resources. To address this issue, HTTP/1.1 introduced a feature called pipelining, allowing a client to send multiple requests without waiting for the server's responses. The server, in turn, responds to the client in the same order as it received the requests.
While pipelining appeared to be a solution, it faced challenges:

- **Server Ignorance or Response Corruption:** Some servers either ignored pipelined requests or corrupted the responses, leading to unreliable communication.
- **Head-of-Line Blocking:** The first request in the pipeline could block subsequent requests, causing a delay in the processing of other requests. This phenomenon, known as head-of-line blocking, resulted in slower page loading times.

In an effort to optimize page loading from servers supporting HTTP/1.1, web browsers implemented a workaround. They open six to eight parallel connections to the server, enabling the simultaneous transmission of multiple requests. This parallelism aims to mitigate the issues associated with pipelining and improve overall page load times.
The choice of six to eight parallel connections by web browsers is based on optimization considerations. The specific reasons behind selecting this number may involve a trade-off between resource utilization, network efficiency, and avoiding potential bottlenecks.

HTTP 2.0

In response to the constraints encountered in pipelining, HTTP/2 introduced a feature called multiplexing. **Multiplexing** allows for **more efficient communication** between the client and server by enabling the **concurrent transmission** of **multiple requests** and responses **over a single connection**.
HTTP/2 utilizes a binary framing mechanism, which means that HTTP messages are broken down into smaller, independent units called frames. These frames can be interleaved and sent over the connection independently of one another. At the receiving end, the frames are reassembled to reconstruct the original HTTP message.
This binary framing mechanism is fundamental to achieving multiplexing in HTTP/2. It enables the browser to send multiple requests over the same connection without encountering blocking issues. As a result, browsers like Chrome utilize the same connection ID for HTTP/2 requests, allowing for efficient and uninterrupted communication between the client and server.
In essence, HTTP/2's multiplexing feature, enabled by the binary framing mechanism, enhances the efficiency and speed of data exchange between clients and servers by facilitating concurrent transmission of multiple requests and responses over a single connection.

TsgcHTTPServer

To improve the performance of the HTTP/2 protocol, the requests are dispatched by default in a pool of threads (by default 32) every time a new HTTP/2 request is received by the server. This avoids waits when a single connection sends many concurrent requests that would require sequential processing (in the context of the connection thread) in the absence of this pool of threads.
The behavior of the pool of threads can be configured with the following properties.

- **HTTP2Options.PoolOfThreads.Enabled:** (by default false) enable this to dispatch HTTP/2 requests in the pool of threads instead of the connection thread.
- **HTTP2Options.Threads:** (by default 32) the number of threads used to handle HTTP/2 requests. Set a number according to the number of processors on your server.

To **fine-tune the requests**, selecting which requests must be processed in the pool of threads (because they are time-consuming) while others can be processed in the connection thread, you can use the event **OnHttp2BeforeAsyncRequest**. This event is raised before queuing the request in the pool of threads. Use the parameter **Async** to set whether the request is threaded or not.

```
void __fastcall TForm1::OnHTTP2BeforeAsyncRequest(TObject *Sender, TsgcWSConnection *Connection, const TIdHTTPRec
{
    if (ARequestInfo.Document == "/fast-request")
```

```
        Async = false;
}
```

TsgcHTTPServer | 404 Error without Response Body

By default, the Indy library adds a content body to HTTP responses if there is no ContentText or ContentStream assigned. If you want to return an empty response body (for a 404 error or similar), you can use the following approach.
Create a new TStringStream without content and assign it to the ContentStream property of the HTTP Response. This way the HTTP response will be sent without the default HTML tags.
Example

```
private void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  AResponseInfo->ContentStream = new TStringStream("");
  AResponseInfo->ContentType = "text/html";
  AResponseInfo->ResponseNo = 404;
}
```

TsgcHTTPServer | Sessions

HTTP is a stateless protocol (at least up to HTTP 1.1), so the client requests a file, the server sends a response, and the connection is closed (you can enable keep-alive so the connection is not closed immediately, but that is beyond the scope of this article). Sessions allow you to store information about the client, which can be used during a client login for example. You can use any unique session ID, search the list of sessions to see if one already exists, and if not, create a new session. A session can be destroyed after a period of inactivity or manually after client logout.

Configuration

There are some properties in TsgcHTTPServer that enable/disable sessions in the server component. The most important are:

Property	Description
SessionState	This is the first property that must be enabled in order to use Sessions. Without this prop abled, sessions will not work
SessionTimeout	Here you must set a value greater than zero (in milliseconds) for the maximum time a se be active.
AutoStartSession	Sessions can be created automatically (AutoStartSession = true) or manually (AutoStartSe false). If sessions are created automatically, the server will use RemoteIP as a unique ide check if there is an active session stored.
SessionClass	Optional. The class the server uses when it creates a new session. Set it to your own **TIdH** **sion** descendant to store your own data inside every session. Must be set before the ser tivated.
SessionList	The list that holds the active sessions. Read it to search, create or remove sessions by co can also assign your own list if you need full control over where sessions are stored. Mus signed before the server is activated.

```
TsgcHTTPServer1->SessionState = true;
TsgcHTTPServer1->SessionTimeout = 600000;
TsgcHTTPServer1->AutoStartSession = false;
```

Create Session

To create a new session, you must create a new **session ID** that is **unique**. You can use any value. **Example:** if the client is authenticating, you can use user + password + remoteip as the session ID.
Then, search the session list to check if it already exists. If it does not exist, create a new one.
When a new session is created **OnSessionStart** event is called and when session is closed, **OnSessionEnd** event is raised.

```
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Document == "/")
  {
    AResponseInfo->ServeFile(AContext, "yourpathhere\\index.html");
  }
  else
  {
    // check if user is valid
    if (((ARequestInfo->AuthUsername == "user") && (ARequestInfo->AuthPassword == "pass")) == false)
    {
      AResponseInfo->AuthRealm = "Authenticate";
    }
    else
    {
      // create a new session id with authentication data
      String vID = ARequestInfo->AuthUsername + "_" + ARequestInfo->AuthPassword + "_" + ARequestInfo->RemoteIP;
```

```
      // search session
      TIdHTTPSession *oSession = TsgcHTTPServer1->SessionList->GetSession(vID, ARequestInfo->RemoteIP);

      // create new session if not exists
      if (oSession == NULL)
      {
        oSession = TsgcHTTPServer1->SessionList->CreateSession(ARequestInfo->RemoteIP, vID);
      }

      AResponseInfo->ContentText = "<html><head></head><body>Authenticated</body></html>";
      AResponseInfo->ResponseNo = 200;
    }
  }
}
```

Read the Current Session

Once a session exists, the server attaches it to every request that carries the session cookie. Read it from **ARequestInfo.Session**, which is nil when the request has no session.

```
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Session != NULL)
  {
    // Content is a TStrings you can use to store your own values
    int vVisits = StrToIntDef(ARequestInfo->Session->Content->Values["visits"], 0) + 1;
    ARequestInfo->Session->Content->Values["visits"] = IntToStr(vVisits);

    AResponseInfo->ContentText = "Session " + ARequestInfo->Session->SessionID +
      " visits: " + IntToStr(vVisits);
  }
  else
  {
    AResponseInfo->ContentText = "No session";
  }
}
```

Use Your Own Session Class

If you want to keep your own fields inside every session, instead of using the **Content** string list, create a descendant of **TIdHTTPSession** and tell the server to use it through the **SessionClass** property. Set it **before** the server is activated.
The server keeps taking care of everything else: it generates the unique session ID, it sends the session cookie, it applies SessionTimeout and it removes stale sessions. Override the virtual constructor **CreateInitialized** if you want to initialize your own fields when the session is created.

```
class TMySession : public TIdHTTPSession
{
private:
  String FUserName;
  TDateTime FLoginTime;
public:
  __fastcall virtual TMySession(TIdHTTPCustomSessionList *AOwner,
    const String SessionID, const String RemoteIP)
    : TIdHTTPSession(AOwner, SessionID, RemoteIP)
  {
    FLoginTime = Now();
  }
  __property String UserName = {read = FUserName, write = FUserName};
  __property TDateTime LoginTime = {read = FLoginTime, write = FLoginTime};
};

// configure the server before it starts
TsgcHTTPServer1->SessionState = true;
TsgcHTTPServer1->SessionTimeout = 600000;
TsgcHTTPServer1->SessionClass = __classid(TMySession);
TsgcHTTPServer1->Active = true;

// and read it back in any request
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
  TIdHTTPResponseInfo *AResponseInfo)
{
  TMySession *oSession = dynamic_cast<TMySession *>(ARequestInfo->Session);
  if (oSession != NULL)
  {
```

```
    AResponseInfo->ContentText = oSession->UserName;
  }
}
```

Use Your Own Session List

If you need full control over how sessions are stored, for example keeping them in a database or sharing them between several servers, assign your own list to the **SessionList** property, before the server is activated. Descend from **TIdHTTPDefaultSessionList** and override the virtual **CreateSession** method. The server calls it internally from CreateUniqueSession, so the unique session ID is still generated for you.
When you assign your own SessionList, the **SessionClass** property no longer applies, because your list decides which class to create.

```
class TMySessionList : public TIdHTTPDefaultSessionList
{
public:
  virtual TIdHTTPSession * __fastcall CreateSession(const String RemoteIP,
    const String SessionID)
  {
    TIdHTTPSession *oSession = new TMySession(this, SessionID, RemoteIP);
    SessionList->Add(oSession);
    return oSession;
  }
};

// assign it before the server starts
TsgcHTTPServer1->SessionList = new TMySessionList(NULL);
TsgcHTTPServer1->Active = true;
```

A Note About OnCreateSession

The OnCreateSession event lets you return a session instance yourself, but it does **not** assign a session ID to it. If you create the session with a plain constructor, the session ID and the session cookie are empty and the session can never be found again on the next request. Use **SessionClass** instead, which is simpler and handles all of that for you.
Also, do not call **SessionList.CreateUniqueSession** inside OnCreateSession. That method already adds the new session to the list, and the server adds it again when your handler returns, so the same session ends up twice in the list.

TsgcHTTPServer | Stream Video

If you want to stream a video file using the server, you can use the function IndyStreamFileVideo to stream the file using chunked transfer encoding.
The function takes the following parameters:

- **AContext:** it's taken from OnCommandGet event.
- **AResponseInfo:** it's taken from OnCommandGet event.
- **aFileName:** it's the full filename of the video to stream.
- **ContentType:** by default is "video/mpeg".
- **aBufferSize:** by default is 1024 bytes.

Example: stream a video when a user goes to the document /video.mp4 and this video is in the folder c:\videos\video.mp4

```
void OnServerCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo *AResponseInf
{
  if (ARequestInfo->Document == "/video.mp4") {
    IndyStreamFileVideo(Acontext, AResponseInfo, "C:\\videos\\video.mp4");
  } else {
    AResponseInfo->ResponseNo = 404;
  }
}
```

Server SSL | SChannel for Indy Servers

Indy-based server components (**TsgcWebSocketServer**, **TsgcWebSocketHTTPServer**, **TsgcHTTPServer**) can use **Windows SChannel** (Secure Channel) as the TLS provider instead of OpenSSL. SChannel is the native Windows TLS implementation, so it does not require any external DLLs.

How it works

When a server component has SSL enabled and the IOHandler is set to **iohSChannel**, the server creates a **TsgcIdServerIOHandlerSSLSChannel** instance that handles all TLS operations using the Windows SChannel API. For every incoming client connection the server performs the TLS handshake through the SChannel provider, negotiating the protocol version, cipher suite, and binding the configured certificate.
SChannel reads certificates from the **Windows Certificate Store** or from a **PFX file** (.pfx / .p12). No PEM files are needed and no OpenSSL libraries need to be deployed.

Configuration

To enable SChannel on an Indy server, configure the following properties:
1. Set the **SSL** property to **True**.
2. Set **SSLOptions.IOHandler** to **iohSChannel**.
3. Set **SSLOptions.Version** to the desired TLS version (tls1_2, tls1_3, ...).
4. Set **SSLOptions.Port** to the port used for secure connections.
5. Configure the certificate using one of the two methods described below.

SChannel_Options properties

The **SSLOptions.SChannel_Options** sub-property exposes the SChannel-specific settings:

Property	Description
CertHash	The thumbprint (hexadecimal hash) of a certificate installed in the Windows Certificate Store.
CertStoreName	Which certificate store to search: **scsnMY** (Personal), scsnRoot, scsnTrust, scsnCA.
CertStorePath	Store location: **scspStoreLocalMachine** or scspStoreCurrentUser.
CipherList	Optional colon-separated list of cipher algorithms (e.g. CALG_AES_256:CALG_AES_128). Empty means use system defaults.
UseLegacyCredentials	When True, uses the legacy SCHANNEL_CRED structure instead of SCH_CREDENTIALS. Enable this for older Windows versions that do not support the newer API.

Additionally, the general **SSLOptions** properties **CertFile** and **Password** are used when loading a certificate from a PFX file.

Certificate from Windows Store

If the certificate is already installed in the Windows Certificate Store, provide the certificate **thumbprint** and indicate where it is located.
To find the thumbprint, open **PowerShell** and run:

```
dir cert:\localmachine\my
```

The **Thumbprint** column shows the hexadecimal hash you need.

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint                                Subject
----------                                -------
C12A8FC8AE668F866B48F23E753C93D357E9BE10  CN=*.mydomain.com
```

```
oServer = new TsgcHTTPServer();
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->Version = tls1_2;
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->SSLOptions->SChannel_Options->CertHash = "C12A8FC8AE668F866B48F23E753C93D357E9BE10";
oServer->SSLOptions->SChannel_Options->CertStoreName = scsnMY;
oServer->SSLOptions->SChannel_Options->CertStorePath = scspStoreLocalMachine;
oServer->Active = true;
```

Certificate from PFX file

If you have a PFX (.pfx or .p12) certificate file, set the **CertFile** and **Password** properties on SSLOptions. SChannel will import the certificate at startup.

```
oServer = new TsgcHTTPServer();
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->Version = tls1_2;
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->SSLOptions->CertFile = "c:\\certificates\\server.pfx";
oServer->SSLOptions->Password = "mypassword";
oServer->Active = true;
```

If you have a PEM certificate and private key, convert them to PFX format first using OpenSSL:

```
openssl pkcs12 -inkey server.key -in server.crt -export -out server.pfx
```

TLS Version

Use the **SSLOptions.Version** property to control which TLS version the server accepts:

Value	Description
tls1_0	TLS 1.0 (not recommended)
tls1_1	TLS 1.1
tls1_2	TLS 1.2 (recommended)
tls1_3	TLS 1.3
tlsUndefined	Accept TLS 1.0, 1.1 and 1.2

Cipher List

By default SChannel uses the system cipher configuration. You can restrict the allowed ciphers by setting **SChannel_Options.CipherList** to a colon-separated list of algorithm names, for example:

```
CALG_AES_256:CALG_AES_128
```

Leave this property empty to use the Windows defaults.

Legacy Credentials

Windows Server 2019 and earlier may not support the newer **SCH_CREDENTIALS** API. If the server fails to start on an older Windows version, set **SChannel_Options.UseLegacyCredentials** to **True** to use the legacy **SCHANNEL_CRED** structure instead.

The component detects the Windows version automatically in most cases, but you can force legacy mode if needed.

Advantages over OpenSSL

- **No external DLLs:** SChannel is built into Windows, so you do not need to deploy libeay32.dll / ssleay32.dll or libcrypto / libssl.
- **Windows Certificate Store integration:** use certificates already installed and managed by the operating system.
- **Automatic updates:** TLS improvements and security patches are applied through Windows Update.

Notes

- SChannel is available on **Windows only**. For cross-platform servers, use OpenSSL (iohOpenSSL).
- The server must have the **private key** associated with the certificate. When using the Windows Store method, the certificate must have been imported with its private key.
- For production servers using the Certificate Store method, the **Local Machine** store (scspStoreLocalMachine) is recommended so the certificate is available regardless of which user account runs the service.
- Only **PFX** (.pfx / .p12) certificate files are supported. If you have PEM files, convert them to PFX format first.

TsgcHTTPRESTServer

HTTP server component with CORS, the /metrics and /health endpoints and multi-tenancy — TsgcHTTPServer plus the extras a REST service usually needs.

Introduction

TsgcHTTPRESTServer is a TsgcHTTPServer with three extras: CORS, the /metrics and /health endpoints, and multi-tenancy. It descends directly from TsgcHTTPServer, so it inherits the whole HTTP server (Port, Bindings, Active, DocumentRoot, SSL, SSLOptions, Authentication, KeepAlive, HTTPCompression, sessions, HTTP/2, HTTP/3, thread pool, IOCP/EPOLL engines, firewall, rate limiter, OnCommandGet, OnCommandOther, OnBeforeCommand...) and adds exactly four properties on top of it. You find it in the **SGC REST** page of the component palette.
TsgcHTTPServer stays a plain HTTP server and everything extra lives here. That split is deliberate: an application that only publishes an HTTP endpoint keeps the smaller component, and an application that publishes a REST API gets the extras without having to write them again. Because CORS, the statistics endpoints and multi-tenancy are additions and not changes, code written for TsgcHTTPServer keeps working unchanged when the component is swapped.
Each extra is opt in and costs nothing until it is configured. CORS is disabled by default, **ServerStats** and **Tenancy** are unassigned by default, and the tenant resolution returns immediately when no component is attached or its Resolution is trNone, so an unused Tenancy costs a single pointer test per request. A server that started sending Access-Control-Allow-Origin: * after an upgrade would be a security regression, which is why none of the three is ever on unless you turn it on.
Follow the steps below to configure this component:
1. Drop a TsgcHTTPRESTServer component in the form
2. Set Port (default is 80). If you are behind a firewall probably you will need to configure it.
3. Handle OnCommandGet / OnCommandOther to answer requests, or set DocumentRoot to serve static files automatically.

```
AResponseInfo.ContentType := 'application/json';
AResponseInfo.ContentText := '{"status":"ok"}';
```

CORSOptions: set Enabled = true to answer cross-origin browser calls. The OPTIONS preflight is answered by the server with a 204 plus the CORS headers, so you never write an OPTIONS branch of your own. The headers are added once per request and only when the request was not already answered by an API (the OpenAPI engine adds its own), so they are never emitted twice.

ServerStats: assign a TsgcHTTPServerStats component to publish /metrics and /health. Both endpoints are served after the Authentication gate, so they inherit whatever authentication the server enforces and are never public unless the server is.

Tenancy: assign a TsgcHTTPServer_Tenancy component to resolve the tenant of every request from the host name, the path, a header or a JWT claim. Read the resolved value from the **Tenant** property inside OnBeforeCommand, OnCommandGet and OnCommandOther.

4. Create a procedure and set property Active = true.

Reference

- Properties — 4 properties added by this component.
- TsgcHTTPServer — the base component, where all the inherited properties, methods and events are documented.

TsgcHTTPServer_Users

User store component that keeps usernames, salted password hashes and roles, and resolves the credentials of any sgcWebSockets HTTP server assigned to its Authentication.Users property.

Introduction

TsgcHTTPServer_Users is a stand-alone user store. It holds a list of accounts, each one with an Id, a username, a password hash, a per-account random salt, a comma separated list of roles, an enabled flag, the creation and last login timestamps and a free metadata field, and it answers one question: are these credentials valid, and what roles does this account hold. It does not listen on a socket and it does not parse HTTP.
Assign it to the `Authentication.Users` property of a server, for example TsgcHTTPRESTServer, and the built-in HTTP Basic gate resolves every incoming credential against this store by itself. No `OnAuthentication` handler is required. The same `Authentication.Users` property is published by the other sgcWebSockets HTTP servers, such as TsgcHTTPServer and TsgcWebSocketHTTPServer, so a single store can serve several endpoints.
Passwords are never kept in clear text. Every account gets its own random salt, the password is hashed with SHA-512 (or SHA-256) over the configured number of iterations, and the candidate hash is compared in constant time so a wrong password cannot be discovered by timing the answer.
Follow the steps below to configure this component:
1. Drop a TsgcHTTPServer_Users component in the form.
2. Choose where the accounts live with Storage. Leave `StorageType = ustMemory` for a store that lives only while the application runs, set `ustFile` and a `FileName` to persist it to disk, or set `ustCustom` to keep the accounts in your own database.
3. Optionally protect the file at rest with `Storage.EncryptAtRest` and `Storage.EncryptionKey`, and tune Hashing: the `Algorithm`, the number of `Iterations` and the optional pepper in `Hashing.Salt`.
4. Create the accounts with AddUser and grant permissions with AddRole.

```
oUsers.AddUser('admin', 'secret', 'admin,reader');
```

5. Assign the store to the server and enable authentication.

```
oServer.Authentication.Users := oUsers;
oServer.Authentication.Enabled := True;
oServer.Active := True;
```

* The server resolves an incoming HTTP Basic credential in a fixed order: the `Authentication.AuthUsers` list first, so an existing deployment behaves exactly as it did, then this user store, and finally the `OnAuthentication` event, which stays the last word.

* A disabled account is rejected by the store itself, so nothing has to be checked again in the server.

Reference

- Properties — 3 properties available on the component.
- Methods — 22 public methods.
- Events — 5 events you can handle.

TsgcHTTPServer_Tenancy

Multi-tenancy resolver component — maps the host name, the request path, a request header or a JWT claim of every request to a tenant name.

Introduction

TsgcHTTPServer_Tenancy is a pure tenant resolver: it knows nothing about Indy, sockets or the server, it only maps the four inputs of a request (the host name, the request path, the value of a request header and the payload of the bearer JWT) to a tenant name. Multi-tenancy means that a single server process serves several customers, the tenants, which share the same endpoints but never the same data. The component answers one question only, which tenant does this request belong to, and leaves the routing, the data isolation and the validation to the application.
Attach the component to the Tenancy property of a TsgcHTTPRESTServer and the server resolves the tenant of every request before any user code runs. The wiring is a single line:

```
oRESTServer.Tenancy := oTenancy;
```

The resolved value is then read back from the read-only Tenant property of the server. Tenant is thread local, so every request thread sees its own value, and it is valid inside OnBeforeCommand, OnCommandGet and OnCommandOther.

Resolution returns immediately when Resolution is trNone, so an unused Tenancy costs a single pointer test per request. These are the five resolution modes:

Mode	Source	Configured with
`trNone`	Resolution is off, the server request path is untouched.	—
`trHost`	From the host name.	HostSuffix
`trPath`	From the request path.	PathSegmentIndex
`trHeader`	From a request header.	HeaderName
`trJWTClaim`	From a claim of the bearer JWT payload.	ClaimName

Follow the steps below to configure this component:

1. Drop a TsgcHTTPServer_Tenancy component in the form.

2. Set Resolution to the mode you need. The default is trNone, which is a hard off switch: no built-in resolution, no event and no default tenant, so an attached component changes nothing at all.

3. Configure the property that belongs to the selected mode: HostSuffix for trHost, PathSegmentIndex for trPath, HeaderName for trHeader (default `X-Tenant-Id`) or ClaimName for trJWTClaim (default `tenant`).

4. Optionally set DefaultTenant, which is returned when the configured resolution yields nothing, and handle OnResolveTenant when you want to override the built-in resolution with your own rule.

5. Assign the component to the Tenancy property of a TsgcHTTPRESTServer and read the resolved value from the server Tenant property inside OnBeforeCommand, OnCommandGet or OnCommandOther.

With trHost and HostSuffix `.example.com`, a request to `acme.example.com` resolves to `acme`. With trPath and PathSegmentIndex `0`, a request to `/acme/api/orders` resolves to `acme`.

* The resolved value comes from the request and is therefore attacker controlled. Never use it directly as a file name, a directory or an SQL identifier without validating it against the list of tenants the application knows.

Reference

- Properties — 6 properties available on the component.
- Methods — 2 public methods.
- Events — 1 event you can handle.

TsgcHTTPServerStats

Statistics component for the REST server — counts requests, responses by status class, latency and connections, and publishes them on /metrics in Prometheus format and on /health as JSON.

Introduction

TsgcHTTPServerStats is the single place to ask how a server is doing. It keeps one set of counters per component instance: total requests, responses grouped by status class (1xx, 2xx, 3xx, 4xx, 5xx), the minimum, mean and maximum request duration in milliseconds, the number of connections currently open, the seconds elapsed since the counters started, and one request counter per request document. Every counter is updated under a critical section, so the component is safe to read from any thread while the server is serving.
The component does not intercept anything by itself: it only counts and formats. You attach it to a server the same way the firewall or the rate limiter are attached, by assigning it to the **ServerStats** property of a TsgcHTTPRESTServer. From that moment the server feeds it with every request it serves and answers the two built-in endpoints on its behalf.
Both endpoints are **disabled by default** and must be enabled one by one: `Endpoints.Metrics.Enabled` and `Endpoints.Health.Enabled` are both False when the component is created, so a server that is upgraded never starts publishing statistics on its own. The paths default to `/metrics` and `/health` and can be changed to anything you prefer. Both endpoints are served after the server Authentication gate, so they inherit whatever authentication the server enforces (Basic, JWT, OAuth2, API key) and are never public unless the server itself is public.
The /metrics endpoint answers the Prometheus text exposition format version 0.0.4 with the content type `text/plain; version=0.0.4; charset=utf-8`, so a Prometheus server scrapes it directly with **no exporter, no sidecar and no bridge in between**. Point a scrape job at the endpoint and the series show up as they are.
Follow the steps below to configure this component:
1. Drop a TsgcHTTPServerStats component in the form, or create it by code.

2. Enable the endpoints you want to publish. Both are off by default and each one is enabled individually.

```
oStats.Endpoints.Metrics.Enabled := true;
oStats.Endpoints.Health.Enabled := true;
```

3. Change the paths only if the defaults clash with your own routes. An empty Path disables the endpoint even when Enabled is true.

```
oStats.Endpoints.Metrics.Path := '/internal/metrics';
oStats.Endpoints.Health.Path := '/internal/health';
```

4. Assign the component to the ServerStats property of a TsgcHTTPRESTServer. That single line is the whole wiring.

```
oServer.ServerStats := oStats;
```

5. Optionally attach the other server components you already use. When a **RateLimiter**, a **Firewall**, an **APIKeyManager** or a **CircuitBreaker** is assigned, its own counters are appended to the metrics output as extra `sgc_ratelimiter_*`, `sgc_firewall_*`, `sgc_apikey_*` and `sgc_circuitbreaker_*` blocks. Those four properties are compiled only when the matching conditional (`SGC_RATELIMITER`, `SGC_FIREWALL`, `SGC_APIKEYMANAGER`, `SGC_CIRCUITBREAKER`) is defined, and the counters are read from the components, never duplicated here.

6. Set the server property Active = true and scrape the endpoint.

/metrics output: Prometheus text exposition format 0.0.4, LF line endings. The metric names below are always emitted; the per-endpoint block is added for every distinct request document that has been counted.

```
# HELP sgc_server_uptime_seconds Seconds since the counters started.
# TYPE sgc_server_uptime_seconds gauge
sgc_server_uptime_seconds 3600
# HELP sgc_server_active_connections Connections currently open on the server.
# TYPE sgc_server_active_connections gauge
sgc_server_active_connections 12
# HELP sgc_server_requests_total HTTP requests handled by the server.
# TYPE sgc_server_requests_total counter
sgc_server_requests_total 48219
```

```
# HELP sgc_server_responses_total HTTP responses by status class.
# TYPE sgc_server_responses_total counter
sgc_server_responses_total{class="1xx"} 0
sgc_server_responses_total{class="2xx"} 47980
sgc_server_responses_total{class="3xx"} 12
sgc_server_responses_total{class="4xx"} 214
sgc_server_responses_total{class="5xx"} 13
# HELP sgc_server_request_duration_ms_min Fastest request served, in millisec-
onds.
# TYPE sgc_server_request_duration_ms_min gauge
sgc_server_request_duration_ms_min 0
# HELP sgc_server_request_duration_ms_avg Mean request duration, in millisec-
onds.
# TYPE sgc_server_request_duration_ms_avg gauge
sgc_server_request_duration_ms_avg 7
# HELP sgc_server_request_duration_ms_max Slowest request served, in millisec-
onds.
# TYPE sgc_server_request_duration_ms_max gauge
sgc_server_request_duration_ms_max 412
# HELP sgc_server_endpoint_requests_total HTTP requests by request document.
# TYPE sgc_server_endpoint_requests_total counter
sgc_server_endpoint_requests_total{endpoint="/api/orders"} 31004
sgc_server_endpoint_requests_total{endpoint="/api/users"} 17215
```

Endpoint cardinality: the per-endpoint counter keeps at most 256 distinct paths in memory. Once that cap is reached every new path is counted under the literal `other`, so a client cannot grow the list without limit just by asking for random documents. Paths are lowercased before they are counted and an empty document is counted as `/`.

/health output: a single JSON object with the content type `application/json`. `status` is `ok`, or `degraded` when a CircuitBreaker is assigned and it currently has open breakers.

```
{"status":"ok","uptime":3600,"connections":12,"requests":48219,"responses":
{"1xx":0,"2xx":47980,"3xx":12,"4xx":214,"5xx":13},"latency":
{"min":0,"avg":7,"max":412}}
```

Active: collection is on as soon as the component is attached. Set Active = false to suspend the counters without detaching the component; the two endpoints stop answering as well, because both request tests require Active.

Reference

- Properties — 13 properties available on the component.
- Methods — 8 public methods.
- Events — 1 event you can handle.
- TsgcHTTPRESTServer — the server component this one is attached to, through its ServerStats property.

TsgcWebSocketServer_HTTPAPI

The HTTP Server API enables applications to communicate over HTTP without using Microsoft Internet Information Server (IIS).

Introduction

The HTTP Server API enables applications to communicate over HTTP without using Microsoft Internet Information Server (IIS). Applications can register to receive HTTP requests for particular URLs, receive WebSocket requests, and send WebSocket responses. The HTTP Server API includes SSL support so that applications can exchange data over secure HTTP connections without IIS. It is also designed to work with I/O completion ports.

The server supports the following protocols:

- WebSockets (Requires Windows 8 or later)
- HTTP 1.1
- HTTP/2 (Requires Windows 2016+ or Windows 10+).

By default, this component requires that your application run as Administrator mode, for URL registration. If the URL has already been registered using an external tool like netsh, you can run without Admin rights, disable the property BindingOptions.ConfigureSSLCertificate to allow starting the application without admin rights.
Set FastMM4/FastMM5 as the first unit of your project.

Follow the steps below to configure this component:

1. Drop a TsgcWebSocketServer_HTTPAPI component in the form

2. Define the listening address and port:

```
Server->Host = "127.0.0.1";
Server->Port = 80;
```

3. Set Specifications allowed, by default, all specifications are allowed.

RFC6455: is standard and recommended WebSocket specification.

Hixie76: it's a draft and it's only recommended to establish Hixie76 connections if you want to provide support to old browsers like Safari 4.2

5. Create a procedure and set property Active = true

URL Reservation

The HTTP.SYS server uses URL reservation to assign which URL endpoints will be used by the HTTP.SYS server.

Basic URL Reservation

This is the most easy simple mode to configure the Server, basically you only set the Host and Port that the HTTP.SYS server will handle.
Example: if your server runs on the IP 127.0.0.1 and Port 80, just set the following properties

```
Server->Host = "127.0.0.1";
Server->Port = 80;
```

If the server runs in more than one IP and you want bind to multiple IPS, use the **NewBinding** Method. First clear the Host and Bindings property and then use the NewBinding method to define all Server Bindings.

```
Server->Host = "";
Server->Bindings->Clear;
Server->Bindings->NewBinding("127.0.0.1", 80, "");
Server->Bindings->NewBinding("80.50.55.11", 80, "");
```

If the server requires SSL connections, do the following to define the Host and Port which will be used to handle SSL connections.

```
Server->Host = "127.0.0.1";
Server->Port = 443;
Server->SSL = true;
Server->SSLOptions->Hash = "CERTIFICATE_HASH";
```

If the server requires SSL connections with multiple IP Addresses, first clear the Host and Bindings property and the register the new Bindings.

```
Server->Host = '';
Server->Bindings->Clear;
Server->Bindings->NewBinding("127.0.0.1", 443, "", true, "CERTIFICATE_HASH1");
Server->Bindings->NewBinding("80.50.55.11", 443, "", true, "CERTIFICATE_HASH2");
```

Reference

- Properties — 27 properties available on the component.
- Methods — 9 public methods.
- Events — 23 events you can handle.

Guides

- **Configuration**
  - URL Reservation
- **Connection**
  - OnDisconnect not fired
- **SSL**
  - HTTPAPI Server SSL
  - Self-Signed Certiifcates
- **HTTP**
  - Custom Headers
  - Send Text Response
  - Send File Response
  - Post Big Files
- **HTTP/2**
  - Disable HTTP/2

HTTPAPI | URL Reservation

HTTP.SYS URL reservation is a feature in the Windows operating system that allows a user to reserve a specific Uniform Resource Locator (URL) for their application or service. When a URL is reserved using HTTP.SYS, the operating system will intercept any incoming HTTP requests for that URL and route them to the specified application or service.

To reserve a URL using HTTP.SYS, an application or service must first register the URL with the HTTP.SYS driver by making a call to the HTTP API. The application or service specifies the URL, the HTTP method (e.g., GET, POST), and any additional settings such as authentication requirements.

Once the URL is registered, HTTP.SYS will intercept any incoming HTTP requests for that URL and look up the registered application or service based on the URL and method. If a matching application or service is found, the HTTP.SYS driver will pass the request to that application or service for processing.

NETSH Commands

Register an URL
In this example, the URL http://example.com:80/ is being registered for the user DOMAIN\user. You can replace this with your desired URL and user.

```
netsh http add urlacl url=http://example.com:80/ user=DOMAIN\user
```

Delete an URL
In this example, the URL http://example.com:80/ is being deleted. You can replace this with the URL you want to delete.

```
netsh http delete urlacl url=http://example.com:80/
```

Show All URLs
This command will display a list of all registered URL reservations on the system.

```
netsh http show urlacl
```

TsgcWebSocketServer_HTTPAPI

The HTTP.SYS server registers the URLs automatically when it is started. This is done using the following parameters and methods.

- **Host and Port:** if Host not empty and the Port is different from zero, the server will try to register the URL. Example: the URL https://127.0.0.1:5000 will be registered using the following properties
  - Host = '127.0.0.1';
  - Port = 5000
  - SSL = True
- **NewBinding:** use this method to register one or multiple URLs.
  - Register the url https://127.0.0.1:5000 --> NewBinding('127.0.0.1', 5000, '/', True)
  - Register the url http://+:5000/ws/ --> NewBinding('+', 5000, '/ws/')

The URL registration requires admin privileges in the following cases:

- Port Number is below 1024
- The host is a wildcard "+", instead of an ip address.

If you want to register the port 443 for all IP Addresses of the server and listen only on the endpoint "/ws/" but you don't want to run the server with admin rights, do the following steps:

- Register the URL using netsh
  - netsh http add urlacl url=https://+:443/ws/ user=DOMAIN\user
- Configure the server with the following binding
  - NewBinding('+', 443, '/ws/', True);
- Disable the property ConfigureSSLCertificate
  - TsgcWebSocketServer_HTTPAPI.BindingOptions.ConfigureSSLCertificate = false;
- Configure the SSL Certificate
  - HTTPAPI Server SSL

HTTPAPI | FineTune

TsgcWSServer_HTTPAPI exposes a published property named `FineTune` of type `TsgcServerHTTPAPI_FineTune`. It groups the low-level kernel-mode knobs that influence how the Windows HTTP Server API (http.sys) queues, dispatches and completes requests.

QueueLength

Wraps the `HttpServerQueueLengthProperty` kernel setting. It controls the maximum number of pending requests http.sys will hold in its kernel-mode queue before the server has dequeued them. When the queue is full, the kernel responds to new connections with **503 Service Unavailable** directly, without ever reaching user mode.

Type: ULONG.
Default: 1000 (the Windows kernel default).
Range: up to 65535 requests.
What it improves: raising this value prevents the kernel from rejecting legitimate traffic on bursty workloads - IoT fleets reconnecting after a network blip, market-data broadcast clients reconnecting after a provider reset, or scheduled job workers that all connect on a shared cron tick.

SkipIOCPOnSuccess

This option is not supported and setting it has no effect. The property is kept only for compatibility, so existing forms and code that use it still load.

Type: Boolean.
Default: False.
Why it is not supported: the overlapped I/O of the server relies on a completion packet for every call, even when the call succeeds at once.

OperatingMode

Selects one of two accept/dispatch architectures.

Type: TsgcHTTPAPIOperatingMode = (ompClassic, ompHighPerf).
Default: ompClassic.

- **ompClassic** - single acceptor thread calls `HttpReceiveHttpRequest` synchronously and hand-dispatches each request to the IOCP worker pool via `PostQueuedCompletionStatus`. This is the historical behavior.
- **ompHighPerf** - implements Microsoft's reference HTTP Server High Performance pattern. The acceptor thread is removed. At startup the server pre-posts `ThreadPoolSize x HighPerfAcceptsPerWorker` async receives on the queue-bound IOCP (defaults to 32 x 4 = 128 concurrent outstanding receives). Each worker services its completion inline and re-posts another async receive, maintaining a sliding window until shutdown.

What it improves: ompHighPerf pays off when the server sees either deep single-stream pipelines (large-frame uploads/downloads) or many concurrent clients (hundreds to thousands). The pre-posted receive window absorbs bursts without per-connection allocation, and the inline dispatch removes the acceptor hand-off bottleneck. Leave the default ompClassic for low-traffic APIs and development environments - on light workloads the overhead of maintaining 128 pre-posted contexts costs more than it saves. The mode can only be changed before the server is activated.

HighPerfAcceptsPerWorker

Controls how many async receives each IOCP worker pre-posts when `OperatingMode = ompHighPerf`. The value is ignored in `ompClassic` mode. The total number of concurrent outstanding receives the server maintains equals `ThreadPoolSize x HighPerfAcceptsPerWorker`.

Type: Integer.
Default: 4.
What it improves: a deeper per-worker window lets the server absorb larger bursts of incoming requests without allocating new contexts on the hot path. Raise it for high-concurrency deployments (IoT fleets, market-data distribution, fan-out brokers); the tradeoff is memory - each pre-posted receive holds a request buffer (~16 KB) reserved until it completes. The default 4 is a conservative middle ground validated against the MSDN HP sample.

Example

The following snippet configures an HTTP.sys server for a high-concurrency IoT backend: a large kernel queue to absorb reconnect storms, and HighPerf dispatch with a widened pre-posted receive window.

```
#include "sgcWebSocket_Server_HTTPAPI.hpp"
#include "sgcWebSocket_HTTPAPI_Server.hpp"   // TsgcHTTPAPIOperatingMode
TsgcWSServer_HTTPAPI *oServer = new TsgcWSServer_HTTPAPI(NULL);
oServer->Host = "0.0.0.0";
oServer->Port = 8080;
// absorb 10,000-device reconnect bursts before kernel-level 503
oServer->FineTune->QueueLength = 10000;
// switch from single-acceptor to pre-posted IOCP workers
oServer->FineTune->OperatingMode = ompHighPerf;
// widen the per-worker pre-posted receive window (32 threads * 8 = 256)
oServer->FineTune->HighPerfAcceptsPerWorker = 8;
oServer->Active = true;
```

For a typical internal or low-traffic API, leave every `FineTune` property at its default:

```
TsgcWSServer_HTTPAPI *oServer = new TsgcWSServer_HTTPAPI(NULL);
oServer->Host = "localhost";
oServer->Port = 8080;
// FineTune defaults: QueueLength=1000, SkipIOCPOnSuccess=False,
//                    OperatingMode=ompClassic, HighPerfAcceptsPerWorker=4
oServer->Active = true;
```

TsgcWebSocketServer_HTTPAPI | HTTPAPI Server SSL

The server can be configured to use **SSL Certificates**. In order to get a production server with a server certificate, you must **purchase** a certificate from a **well-known provider**: Namecheap, GoDaddy, Thawte, etc. For **testing purposes** you can use a **self-signed certificate** (check the Demos/Chat example which uses a self-signed certificate). Read the following article How Create a Self-signed certificate.

Once you have your certificate, you must configure the server to specify which certificate to use for encrypting connections.

Certificate Hash

First you need to know the hash of your certificate. Finding the hash of a certificate is as easy in **powershell** as running a **dir** command on the certificates container.

```
dir cert:\localmachine\my
```

The hash is the hexadecimal **Thumbprint** value.

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint                                Subject
----------                                -------
C12A8FC8AE668F866B48F23E753C93D357E9BE10  CN=*.mydomain.com
```

Once you have the Thumbprint value, just set in **TsgcWebSocketServer_HTTPAPI.TLSOptions.Hash** property.

Once you have set the hash, just set TsgcWebSocketServer_HTTPAPI.SSL = true and your server is ready to start.

If you want to register the certificate manually using netsh, use the following command:

```
netsh http add sslcert ipport=<IP>:<PORT> certhash=<THUMBPRINT> appid="{<GUID>}"
```

<IP>: Specifies the local IP address for the binding. Do not use a wildcard binding. Use a valid IP address.
<PORT>: Specifies the port for the binding.
<THUMBPRINT>: The X.509 certificate thumbprint.
<GUID>: A developer-generated GUID to represent the app for informational purposes.

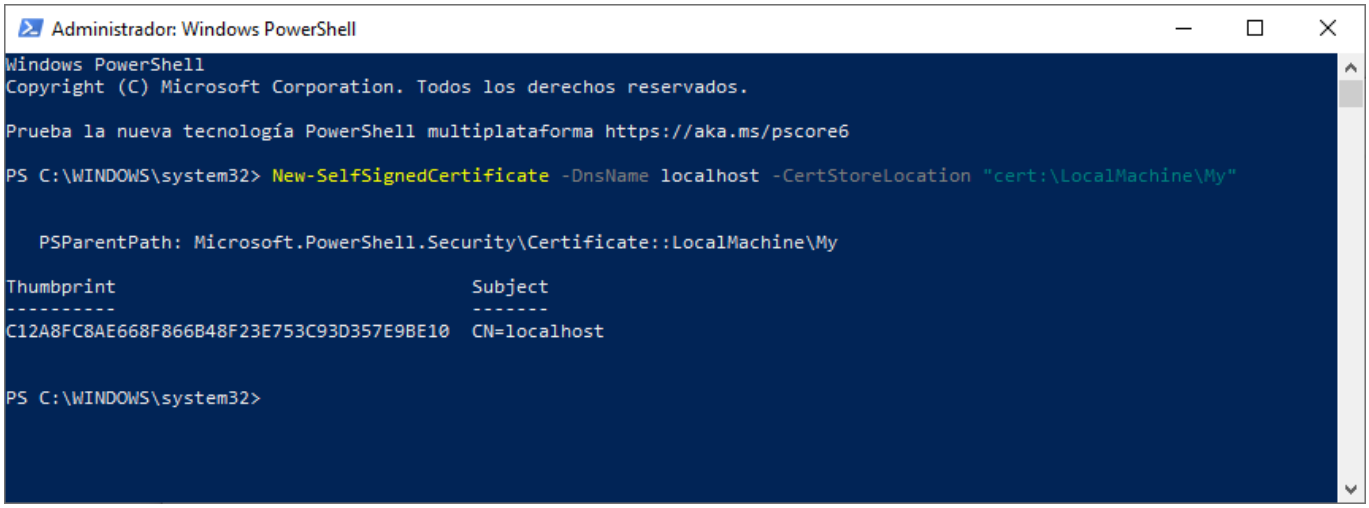
TsgcWebSocketServer_HTTPAPI | Self-Signed Certificates

If you require a certificate for your own testing, you can create a self-signed certificate on your testing machine. Follow these steps:

1. Run **Powershell** as **Administrator**.
2. Run the following command to create the certificate:

```
New-SelfSignedCertificate -DnsName localhost -CertStoreLocation "cert:\LocalMachine\My"
```

If successful, you will get a confirmation about the new certificate created. Just copy Thumbprint and paste on **TsgcWebSocketServer_HTTPAPI.TLSOptions.Hash** property.



3. **Optional**, you can add your self-signed certificate as a **trusted certificate authority**

```
Run MMC -32 as administrator
```

3.1. Select **File / Add or Remove Snap-in**

3.2. Select Certificate and then click Add

3.3. Select computer account and press Next.

3.4. Select Local computer and press Ok. You will now see your Certificates.

4.5. Select your certificate from **Personal / Certificates** and Paste on **Trusted Root Certificates Authorities / Certificates**.

TsgcWebSocketServer_HTTPAPI | Disable HTTP/2

HTTP/2 protocol is enabled by default in **Server 2016+** and **Windows 10+** OS. In some older browsers or HTTP clients, you might encounter an error because the protocol is not fully supported. You can prevent these errors by disabling HTTP/2 protocol.

How to Disable HTTP/2

- Open the Windows Registry Editor
- Go to the following registry key:

  HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\HTTP\Parameters

- Add the following DWORD values and set both values to zero.

  - EnableHttp2Tls
  - EnableHttp2Cleartext

- Reboot the computer.

TsgcWebSocketServer_HTTPAPI | Custom Headers

You can customize the response of HTTP.SYS server using the **CustomHeaders** property of response object.

Set the value of CustomHeaders with the Header Name and Header Value separated by newline characters.

Example: if you want to add the following headers, find below a sample code

```
Access-Control-Allow-Origin: *
Acces-Control-Allow-Methods: GET, POST, OPTIONS, PUT, PATCH, DELETE
```

```
private void OnHTTPRequest(TsgcWSConnection_HTTPAPI *aConnection, const THttpServerRequest *aRequestInfo,
  ref THttpServerResponse *aResponseInfo)
{
  aResponseInfo->ResponseNo = 200;
  aResponseInfo->CustomHeaders = "Access-Control-Allow-Origin: *\r\n" + "Acces-Control-Allow-Methods: " +
    "GET, POST, OPTIONS, PUT, PATCH, DELETE";
}
```

TsgcWebSocketServer_HTTPAPI | Send Text Response

Use the event **OnHTTPRequest** to handle the HTTP Requests.

The class **THttpServerRequest** contains the HTTP Request Data.

- **Document:** the Document the peer is trying to access.
- **Method:** the HTTP Method ('GET', 'POST', etc.)
- **Headers:** the Headers of HTTP request.
- **AcceptEncoding:** accept encoding variable, example: "gzip, deflate, br".
- **ContentType:** example: "text/html"
- **Content:** content of request if exists.
- **QueryParams:** the query parameters.
- **Cookies:** the cookies if exists.
- **ContentLength:** size of the content.
- **AuthExists,** AuthUsername, AuthPassword: authentication request data.
- **Stream:** if the http request has a body, this is the stream of the body.

The class **THttpServerResponse** contains the HTTP response Data.

- **ContentText:** is the response as text.
- **ContentType:** example: "text/html". If you want encode the ContentText with UTF8, set the charset='utf-8'. Example: text/html; charset=utf-8
- **CustomHeaders:** if you need to send your own headers use this variable
- **AuthRealm:** if the server requires authentication, set this variable.
- **ResponseNo:** the HTTP response number. Example: 200 means the response is successful.
- **ContentStream:** if the response contains a stream, set here (don't free the stream, it will be freed automatically).
- **FileName:** if the response is a filename, set here the full path to the filename.
- **Date, Expires, LastModified:** datetime variables of the response.
- **CacheControl:** allows customizing the cache behavior.

Example: if the server receives a GET request for the document "/test.html", send an OK response; otherwise send a 404 if it is a GET request for another document, or error 500 if it is a different method.

```
void __fastcall OnHTTPRequest(TsgcWSConnection_HTTPAPI *aConnection,
        const THttpServerRequest *aRequestInfo,
        THttpServerResponse *aResponseInfo)
{
  if (aRequestInfo->Method == "GET")
  {
    if (aRequestInfo->Document == "/test.html")
        {
          aResponseInfo->ResponseNo = 200;
          aResponseInfo->ContentText = "OK";
          aResponseInfo->ContentType = "text/html; charset=UTF-8";
        }
        else
        {
          aResponseInfo->ResponseNo = 404;
        }
  }
  else
  {
    aResponseInfo->ResponseNo = 500;
  }
}
```

TsgcWebSocketServer_HTTPAPI | Send File Response

Use the FileName property of **THttpServerResponse** object if you want to send a file as a response to an HTTP request.

```
void __fastcall OnHTTPRequest(TsgcWSConnection_HTTPAPI *aConnection,
        const THttpServerRequest *aRequestInfo,
        THttpServerResponse *aResponseInfo)
{
  if (aRequestInfo->Method == "GET")
  {
    if (aRequestInfo->Document == "/test.zip")
        {
          aResponseInfo->ResponseNo = 200;
          aResponseInfo->FileName = "c:\download\test.zip";
          aResponseInfo->ContentType = "application/zip";
        }
        else
        {
          aResponseInfo->ResponseNo = 404;
        }
  }
  else
  {
    aResponseInfo->ResponseNo = 500;
  }
}
```

Resumable Downloads

An HTTP 206 Partial Content response is used when a server is fulfilling a request for a specific portion (range) of a resource, instead of sending the entire file. This is commonly used for resumable downloads, media streaming, and large file transfers.
How it works:

Client Requests a Partial Resource: The client (browser, downloader, or media player) sends a `Range` header specifying the byte range it wants. Example request:

```
GET /video.mp4 HTTP/1.1
Host: example.com
Range: bytes=1000-5000
```

This requests bytes 1000 to 5000 of video.mp4.

Server Responds with HTTP 206: If the server supports range requests, it responds with 206 Partial Content and includes a Content-Range header. Example response:

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 1000-5000/1000000
Content-Length: 4001
Content-Type: video/mp4
```

The Content-Range header shows:

The range served (1000-5000)
The total size of the file (1000000 bytes).
The Content-Length header is the size of the returned portion (4001 bytes).

Client Can Request More Chunks:
The client can send multiple requests for different parts.
This enables resumable downloads and efficient streaming.

```
void __fastcall OnHTTPRequest(TIdContext* AContext, TIdHTTPRequestInfo* ARequestInfo, TIdHTTPResponseInfo* ARespo
{
    std::unique_ptr<TFileStream> oStream(new TFileStream("test.pdf", fmOpenRead | fmShareDenyWrite));
    std::unique_ptr<TIdEntityRangeStrings> oRanges(new TIdEntityRangeStrings(nullptr));
    try {
        oRanges->Text = ARequestInfo->RawHeaders->Values["Range"];
        AResponseInfo->ContentType = "application/pdf";
        if (oRanges->Count > 0) {
            AResponseInfo->ResponseNo = 206; // Partial Content
            AResponseInfo->AcceptRanges = "bytes";
            AResponseInfo->ContentRangeStart = oRanges->Items[0]->StartPos;
            AResponseInfo->ContentRangeEnd = oRanges->Items[0]->EndPos;
            AResponseInfo->ContentRangeInstanceLength = oStream->Size;

            AResponseInfo->ContentStream = new TIdHTTPRangeStream(oStream.release(),
                AResponseInfo->ContentRangeStart, AResponseInfo->ContentRangeEnd);
        } else {
            AResponseInfo->ResponseNo = 200; // OK
            AResponseInfo->ContentStream = oStream.release();
        }
    } catch (...) {
        // Handle exceptions
    }
}
```

TsgcWebSocketServer_HTTPAPI | OnDisconnect not fired

When first working with the HTTPAPI Server, it is very common to see that the OnDisconnect event is not fired immediately when a client closes the connection. The reason is that HTTPAPI Server works a bit differently than other servers like Indy. In the **Indy server** there is **a thread for every connection** and this thread checks every x milliseconds whether the **connection is active**. The **HTTPAPI Server** uses a **thread-pool** that handles all connections and **does not check** for every connection whether it is active or not.

In order to get notified when client closes connection, do the following configuration:

1. If you use a TsgcWebSocketClient, set **Options.CleanDisconnect := True**. This means that before the connection is closed, the client will try to send a notification to the server that the connection will be closed. If the server receives this message, the OnDisconnect event will be called.

2. For other disconnections, the only solution is to write something to the socket; if it fails, the connection is disconnected. **Enable HeartBeat** on HTTPAPI server, and set an interval of 60 seconds for example and a timeout of 0. This configuration means that every 60 seconds all connections will be pinged, and if any are disconnected, **OnDisconnect** event will be fired. You can put a lower value of HeartBeat.Interval, but do not set it too low (1 second for example is too low) because the performance of the server will be affected.

TsgcWebSocketClient_WinHTTP

TsgcWebSocketClient_WinHTTP implements a Client VCL WebSocket Component that can connect to a WebSocket Server.

Introduction

TsgcWebSocketClient_WinHTTP implements a Client VCL WebSocket Component that can connect to a WebSocket Server. It is based on the WinHTTP API and requires Windows 8 or higher. Follow the steps below to configure this component:

1. Drop a TsgcWebSocketClient_WinHTTP component in the form

2. Set Host and Port (default is 80) to connect to an available WebSocket Server. You can set URL property and Host, Port, Parameters... will be updated from URL. Example: wss://127.0.0.1:8080/ws/ will result:

```
oClient = new TsgcWebSocketClient_WinHTTP();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClient->TLS = true;
oClient->Options->Parameters = "/ws/";
```

3. You can select if you want TLS (secure connection) or not, disabled by default.

4. The following events can be used to customize the websocket client flow:

5. Create a procedure and set the Active property to True.

Reference

- Properties — 17 properties available on the component.
- Methods — 8 public methods.
- Events — 10 events you can handle.

TsgcWebSocketFirewall

TsgcWebSocketFirewall implements a comprehensive firewall component that protects WebSocket servers against a wide range of security threats.

Introduction

TsgcWebSocketFirewall implements a comprehensive firewall component that protects WebSocket servers against a wide range of security threats. It provides sixteen protection modules including IP blacklist/whitelist filtering, brute force protection with automatic IP banning, SQL injection detection, XSS (Cross-Site Scripting) detection, connection rate limiting, message flood protection, GeoIP country-based filtering, IP-based bot detection, dynamic threat scoring, path traversal detection, command injection detection, payload size limiting, WebSocket-specific protections (origin validation, frame size limits, subprotocol filtering), progressive ban escalation, a custom rules engine, and real-time statistics.

The firewall integrates automatically with server components through the **Firewall** property available on TsgcWebSocketServer, TsgcWebSocketHTTPServer, and TsgcWebSocketServer_HTTPAPI. Once assigned and configured, it intercepts connections and messages without requiring any manual event wiring.

Quick Start

1. Drop a **TsgcWebSocketFirewall** component on the form.

2. Configure the desired protection modules:

Blacklist: Set Enabled to True and add IPs or CIDR ranges to the IPs list (e.g., "10.0.0.0/8", "192.168.1.100").

Whitelist: Set Enabled to True and add trusted IPs. Whitelisted IPs bypass all other checks.

BruteForce: Set Enabled to True. Configure MaxAttempts (default 5), TimeWindowSec (default 60), and BanDurationSec (default 300).

SQLInjection: Set Enabled to True. Detects common SQL injection patterns in messages.

XSS: Set Enabled to True. Detects cross-site scripting patterns in messages.

RateLimit: Set Enabled to True. Limits concurrent connections per IP (default 10 per 60 seconds).

FloodProtection: Set Enabled to True. Limits messages per second per IP (default 100).

PayloadLimit: Set Enabled to True. Blocks messages exceeding MaxSizeBytes (default 1MB).

PathTraversal: Set Enabled to True. Detects directory traversal patterns.

CommandInjection: Set Enabled to True. Detects shell command injection patterns.

ThreatScore: Set Enabled to True. Assigns dynamic scores to IPs based on violations.

BanEscalation: Set Enabled to True. Escalates ban durations for repeat offenders.

GeoIP: Set Enabled to True. Blocks or allows connections by country code.

BotDetection: Set Enabled to True. Classifies connecting IPs as verified crawlers, datacenter hosts, suspected bots, blocklisted or human.

WebSocketProtection: Set Enabled to True. Validates origins, frame sizes, and subprotocols.

CustomRules: Set Enabled to True. Configurable rule engine for complex filtering.

3. Assign the firewall to a server component:

```
sgcWebSocketHTTPServer1.Firewall := sgcWebSocketFirewall1;
```

4. Start the server. The firewall automatically protects all connections and messages.

Properties

Property	Description
Enabled	Enables or disables the firewall. Default: True.
Blacklist	IP blacklist configuration. Contains Enabled (Boolean) and IPs (TStringList) properties. Supports exact IPs and CIDR notation (e.g., "192.168.0.0/16").
Whitelist	IP whitelist configuration. Whitelisted IPs bypass all security checks. Same structure as Blacklist.
BruteForce	Brute force protection. Properties: Enabled, MaxAttempts (default 5), TimeWindowSec (default 60), BanDurationSec (default 300). Automatically bans IPs that exceed the attempt limit.
SQLInjection	SQL injection detection. Properties: Enabled, Action (faDeny/faAllow/faLog), CustomPatterns (TStringList for additional patterns).
XSS	Cross-site scripting detection. Properties: Enabled, Action (faDeny/faAllow/faLog).
RateLimit	Connection rate limiting. Properties: Enabled, MaxConnectionsPerIP (default 10), TimeWindowSec (default 60).
FloodProtection	Message flood protection. Properties: Enabled, MaxMessagesPerSec (default 100), Action (faDeny/faAllow/faLog).
PayloadLimit	Message size limiting. Properties: Enabled, MaxSizeBytes (default 1048576), Action (faDeny/faAllow/faLog).
PathTraversal	Path traversal detection. Properties: Enabled, Action (faDeny/faAllow/faLog).
CommandInjection	Command injection detection. Properties: Enabled, Action (faDeny/faAllow/faLog), CustomPatterns (TStringList for additional patterns).
ThreatScore	Dynamic threat scoring. Properties: Enabled, AutoBanThreshold (default 80), DecayPerHour (default 5), plus per-violation-type weights for fine-tuning score calculations.
BanEscalation	Progressive ban escalation. Properties: Enabled, Levels (TStringList of durations in seconds), ResetAfterSec (default 86400). Each subsequent ban uses the next level duration. A level value of 0 means permanent ban.

GeoIP	Geographic IP filtering. Properties: Enabled, Mode (gmBlockList/gmAllowList), Countries (TStringList of ISO 3166-1 alpha-2 country codes), DatabaseFile (path to a GeoIP CSV database).
BotDetection	IP-based bot detection. Properties: Enabled, KnownBotsFile, CacheDurationSec (default 3600), DNSTimeoutMS (default 2000), plus ReverseDNS, Datacenter and DNSBL sub-objects. Classify-only: classifies IPs as verified crawlers, datacenter hosts, suspected bots, blocklisted or human without blocking.
WebSocket-Protection	WebSocket-specific protections. Properties: Enabled, AllowedOrigins (TStringList), MaxFrameSize (Integer), AllowedSubprotocols (TStringList).
CustomRules	Custom rules engine. Properties: Enabled, Rules (TCollection of TsgcFirewallRuleItem). Each rule defines conditions and actions for complex filtering logic.

Events

OnFiltered: Fired when a connection or message is blocked. The *Allow* parameter (var Boolean) lets you override the firewall decision.

OnViolation: Fired when a security violation is detected. Provides the IP address, violation type, and details for logging purposes.

OnResolveCountry: Fired when GeoIP needs to resolve an IP to a country code. Assign a handler to provide custom country resolution, or load a GeoIP database file for automatic lookups.

OnBotDetected: Fired when bot detection classifies an IP. Provides the IP address, the TsgcBotClassification verdict, and the matched bot name for logging or analytics.

OnResolveBot: Fired when bot detection needs to classify an IP. Assign a handler to supply a custom verdict instead of the built-in resolver.

OnThreatScoreChanged: Fired when an IP's threat score changes. Provides the IP address together with the old and new score values.

Public Methods

Method	Description
IsConnectionAllowed(IP)	Checks if a connection from the given IP is allowed. Called automatically when integrated with a server.
IsMessageAllowed(IP, Message)	Checks if a message from the given IP passes all message filters. Called automatically when integrated with a server.
RegisterConnection(IP)	Registers a new connection for rate limiting tracking. Called automatically.
UnregisterConnection(IP)	Removes a connection from tracking. Called automatically on disconnect.

RegisterFailedAttempt(IP)	Records a failed authentication attempt for brute force detection. Call this from your OnAuthentication event handler when authentication fails.
BanIP(IP, DurationSec)	Manually bans an IP address. DurationSec = 0 means permanent ban.
UnbanIP(IP)	Removes a ban from the specified IP address.
IsBanned(IP)	Returns True if the specified IP is currently banned.
ClearBans	Removes all active bans.
ClearTracking	Resets all internal tracking data (connection counts, attempt logs, message counts).
SaveBansToFile(FileName)	Saves all active bans to a file for persistence across restarts.
LoadBansFromFile(FileName)	Restores bans from a previously saved file.
LoadGeoIPDatabase(FileName)	Loads a GeoIP CSV database. Expected format: start_ip,end_ip,country_code.
LookupCountry(IP)	Returns the 2-letter ISO country code for an IP address.
GetBotClassification(IP)	Returns the TsgcBotClassification for an IP address (cache-first; a cache miss may return bcUnknown and trigger an async lookup).
GetThreatScore(IP)	Returns the current threat score (0-100) for an IP address.
ResetThreatScore(IP)	Resets an IP's threat score to 0.
GetTopThreats(Count)	Returns a list of IPs with the highest threat scores.
IsOriginAllowed(Origin)	Checks if a WebSocket Origin header is allowed by the WebSocketProtection configuration.
IsFrameSizeAllowed(Size)	Checks if a message size is within the limits defined by WebSocketProtection.MaxFrameSize.
IsSubprotocolAllowed(Subprotocol)	Checks if a WebSocket subprotocol is allowed by the WebSocketProtection configuration.
Stats	Public read-only TsgcFirewallStats object with real-time counters for connections blocked, messages filtered, bans issued, threat scores, and per-module hit counts.

Automatic Integration

When assigned to a server's Firewall property, the component automatically:

- **On new connection:** Checks blacklist, whitelist, bans, rate limits, GeoIP country, and evaluates custom rules. Rejects the connection before the OnTCPConnect event if blocked.

- **On message received:** Checks SQL injection, XSS patterns, payload size, path traversal, command injection, flood limits, and evaluates custom rules. Updates threat scores. Disconnects the client if a violation is detected.
- **On disconnect:** Unregisters the connection from internal tracking.

CIDR Support

The blacklist and whitelist support CIDR notation for IP ranges:

- **192.168.1.0/24** — matches 192.168.1.0 to 192.168.1.255
- **10.0.0.0/8** — matches all 10.x.x.x addresses
- **172.16.0.0/16** — matches 172.16.x.x addresses

Example

```
// Configure firewall
sgcWebSocketFirewall1->Blacklist->Enabled = true;
sgcWebSocketFirewall1->Blacklist->IPs->Add("10.0.0.0/8");
sgcWebSocketFirewall1->Whitelist->Enabled = true;
sgcWebSocketFirewall1->Whitelist->IPs->Add("192.168.1.1");
sgcWebSocketFirewall1->BruteForce->Enabled = true;
sgcWebSocketFirewall1->BruteForce->MaxAttempts = 3;
sgcWebSocketFirewall1->BruteForce->BanDurationSec = 600;
sgcWebSocketFirewall1->SQLInjection->Enabled = true;
sgcWebSocketFirewall1->XSS->Enabled = true;
sgcWebSocketFirewall1->RateLimit->Enabled = true;
sgcWebSocketFirewall1->RateLimit->MaxConnectionsPerIP = 5;
sgcWebSocketFirewall1->FloodProtection->Enabled = true;

// GeoIP: block connections from specific countries
sgcWebSocketFirewall1->GeoIP->Enabled = true;
sgcWebSocketFirewall1->GeoIP->Mode = gmBlockList;
sgcWebSocketFirewall1->GeoIP->Countries->Add("CN");
sgcWebSocketFirewall1->GeoIP->Countries->Add("RU");
sgcWebSocketFirewall1->LoadGeoIPDatabase("geoip.csv");

// Threat scoring with auto-ban
sgcWebSocketFirewall1->ThreatScore->Enabled = true;
sgcWebSocketFirewall1->ThreatScore->AutoBanThreshold = 80;

// Progressive ban escalation (5min, 30min, 2hr, 24hr, permanent)
sgcWebSocketFirewall1->BanEscalation->Enabled = true;
sgcWebSocketFirewall1->BanEscalation->Levels->Add("300");
sgcWebSocketFirewall1->BanEscalation->Levels->Add("1800");
sgcWebSocketFirewall1->BanEscalation->Levels->Add("7200");
sgcWebSocketFirewall1->BanEscalation->Levels->Add("86400");
sgcWebSocketFirewall1->BanEscalation->Levels->Add("0");

// Additional content protection
sgcWebSocketFirewall1->PayloadLimit->Enabled = true;
sgcWebSocketFirewall1->PayloadLimit->MaxSizeBytes = 65536;
sgcWebSocketFirewall1->PathTraversal->Enabled = true;
sgcWebSocketFirewall1->CommandInjection->Enabled = true;

// Persistent bans
sgcWebSocketFirewall1->SaveBansToFile("bans.dat");
sgcWebSocketFirewall1->LoadBansFromFile("bans.dat");

// Assign to server
sgcWebSocketHTTPServer1->Firewall = sgcWebSocketFirewall1;

// Optional: Track failed login attempts
void __fastcall TForm1::ServerAuthentication(TsgcWSConnection *Connection,
  String aUser, String aPassword, bool &Authenticated)
{
  Authenticated = (aUser == "admin") && (aPassword == "secret");
  if (!Authenticated)
    sgcWebSocketFirewall1->RegisterFailedAttempt(Connection->IP);
}

// Optional: Handle firewall events for logging
void __fastcall TForm1::FirewallViolation(TObject *Sender, const String aIP,
  const TsgcFirewallViolationType aViolationType, const String aDetails)
```

```
{
  Log("Firewall violation from " + aIP + ": " + aDetails);
}
```

SQL Injection Patterns Detected

The built-in SQL injection detector checks for these patterns (case-insensitive):

- ' OR ', ' AND ' — Boolean injection
- UNION SELECT — Union-based injection
- '; DROP, '; DELETE, '; INSERT, '; UPDATE — Statement injection
- -- (SQL comments)
- EXEC(, EXECUTE(— Command execution
- xp_cmdshell — Extended stored procedures
- CAST(, CONVERT(— Type conversion abuse
- 1=1, 1'='1 — Tautology injection

Additional custom patterns can be added via the SQLInjection.CustomPatterns property.

XSS Patterns Detected

- <script — Script injection
- javascript: — Protocol-based XSS
- onerror=, onload=, onclick=, onmouseover= — Event handler injection
- eval(— JavaScript evaluation
- document.cookie — Cookie theft
- <iframe, <object, <embed — Element injection
- <svg onload — SVG-based XSS
- expression(— CSS expression injection

Path Traversal Patterns Detected

The built-in path traversal detector checks for these patterns:

- ../ and ..\ — Relative directory traversal
- %2e%2e%2f, %2e%2e/, ..%2f — URL-encoded variants
- %00 — Null byte injection
- /etc/passwd — Unix sensitive file access
- c:\windows — Windows system directory access

Command Injection Patterns Detected

The built-in command injection detector checks for these patterns:

- ; | && || — Shell operators (command chaining)
- `command` — Backtick execution
- $(command) — Subshell execution
- rm -rf — Destructive file operations
- wget, curl — Remote file download
- /bin/sh, /bin/bash — Unix shell invocation
- cmd.exe — Windows command interpreter
- powershell — PowerShell invocation

Additional custom patterns can be added via the CommandInjection.CustomPatterns property.

Thread Safety

The firewall component is fully thread-safe. All public methods use internal critical sections to protect concurrent access to tracking data. It has been stress-tested with 20 concurrent threads performing 100,000 operations with zero errors and zero memory leaks.

The component can safely be shared across multiple server instances and accessed from any thread (server event handlers, timer threads, etc.) without external synchronization.

Reference

- Properties — 16 properties available on the component.
- Methods — 21 public methods.
- Events — 6 events you can handle.

Firewall: Blacklist and Whitelist

TsgcWebSocketFirewall provides IP-based access control through two complementary mechanisms: a blacklist that blocks specific IPs and a whitelist that grants unconditional access to trusted IPs.

Blacklist

The blacklist prevents connections from specified IP addresses or IP ranges. When enabled, any incoming connection from a blacklisted IP is rejected before reaching the server's connection events.

Property	Description
Blacklist.Enabled	Enables or disables blacklist checking. Default: False.
Blacklist.IPs	TStringList containing blocked IP addresses or CIDR ranges.

Adding IPs at Design Time

Click the IPs property in the Object Inspector to open the String List editor. Add one IP or CIDR range per line:

```
192.168.1.100
10.0.0.0/8
172.16.0.0/12
```

Adding IPs at Runtime

```
sgcWebSocketFirewall1->Blacklist->Enabled = true;
sgcWebSocketFirewall1->Blacklist->IPs->Add("192.168.1.100");
sgcWebSocketFirewall1->Blacklist->IPs->Add("10.0.0.0/8");
```

Whitelist

The whitelist grants unconditional access to specified IP addresses. **Whitelisted IPs bypass all other firewall checks**, including blacklist, brute force bans, rate limits, and message filtering.

Property	Description
Whitelist.Enabled	Enables or disables whitelist checking. Default: False.
Whitelist.IPs	TStringList containing trusted IP addresses or CIDR ranges.

Example

```
// Allow internal network unconditionally
sgcWebSocketFirewall1->Whitelist->Enabled = true;
sgcWebSocketFirewall1->Whitelist->IPs->Add("192.168.1.0/24");
sgcWebSocketFirewall1->Whitelist->IPs->Add("127.0.0.1");
```

Priority Order

When both blacklist and whitelist are enabled, the firewall evaluates them in this order:
1. If the IP is whitelisted, the connection is **allowed immediately**. No further checks are performed.
2. If the IP is blacklisted, the connection is **denied**.
3. If the IP is in neither list, the connection proceeds to other checks (brute force, rate limiting, etc.).
This means a whitelist entry always takes priority over a blacklist entry for the same IP.
Both lists are matched against the address the firewall holds for the connection, which is the socket address unless ForwardedHeaders resolved a real client address behind a reverse proxy. Note that the connect-time check always runs against the socket peer, so a proxy in front of the server must itself be accepted by these lists. See Firewall: Real Client IP Behind a Reverse Proxy.

CIDR Notation

Both blacklist and whitelist support CIDR (Classless Inter-Domain Routing) notation for specifying IP ranges:

CIDR	Range	Addresses
192.168.1.0/24	192.168.1.0 - 192.168.1.255	256
192.168.0.0/16	192.168.0.0 - 192.168.255.255	65,536
10.0.0.0/8	10.0.0.0 - 10.255.255.255	16,777,216
172.16.0.0/12	172.16.0.0 - 172.31.255.255	1,048,576

You can mix exact IPs and CIDR ranges in the same list:

```
sgcWebSocketFirewall1->Blacklist->IPs->Add("203.0.113.50");     // single IP
sgcWebSocketFirewall1->Blacklist->IPs->Add("198.51.100.0/24");  // entire subnet
```

Combining Blacklist and Whitelist

A common pattern is to block a broad range but allow specific IPs within that range:

```
// Block the entire 10.x.x.x range
sgcWebSocketFirewall1->Blacklist->Enabled = true;
sgcWebSocketFirewall1->Blacklist->IPs->Add("10.0.0.0/8");

// But allow the monitoring server
sgcWebSocketFirewall1->Whitelist->Enabled = true;
sgcWebSocketFirewall1->Whitelist->IPs->Add("10.1.1.50");
```

In this example, all IPs in the 10.x.x.x range are blocked except 10.1.1.50, which is whitelisted and bypasses all checks.

Firewall: Brute Force Protection

The brute force protection module detects repeated failed authentication attempts from the same IP address and automatically bans the offending IP for a configurable duration.

How It Works

The firewall tracks failed attempts per IP address within a sliding time window. When the number of failures exceeds the configured threshold, the IP is automatically banned. Banned IPs are rejected at the connection level, before any server events fire.

1. A client connects and attempts authentication.
2. If authentication fails, your code calls **RegisterFailedAttempt(IP)** to notify the firewall.
3. The firewall increments the failure counter for that IP within the current time window.
4. If the counter reaches **MaxAttempts**, the IP is banned for **BanDurationSec** seconds.
5. Any new connection from a banned IP is rejected immediately.
6. After the ban duration expires, the IP is automatically unbanned and the failure counter resets.

Properties

Property	Default	Description
BruteForce.Enabled	False	Enables or disables brute force protection.
BruteForce.MaxAttempts	5	Maximum number of failed attempts allowed within the time window before the IP is banned.
BruteForce.TimeWindowSec	60	Duration of the sliding time window in seconds. Failed attempts older than this are discarded.
BruteForce.BanDurationSec	300	Duration of the automatic ban in seconds. Set to 0 for a permanent ban (until manually removed or server restart).

RegisterFailedAttempt

The firewall does not know which connections represent failed logins. You must call **RegisterFailedAttempt** from your authentication event handler whenever a login attempt fails:

```
void __fastcall TForm1::ServerAuthentication(TsgcWSConnection *Connection,
  String aUser, String aPassword, bool &Authenticated)
{
  Authenticated = CheckCredentials(aUser, aPassword);
  if (!Authenticated)
    sgcWebSocketFirewall1->RegisterFailedAttempt(Connection->IP);
}
```

Manual Ban Management

In addition to automatic banning, you can manually manage bans:

Method	Description

BanIP(IP, DurationSec)	Manually bans an IP. Use DurationSec = 0 for a permanent ban.
UnbanIP(IP)	Removes a ban from the specified IP.
IsBanned(IP)	Returns True if the IP is currently banned.
ClearBans	Removes all active bans (both automatic and manual).

Example: Manual Ban

```
// Ban an IP for 1 hour
sgcWebSocketFirewall1->BanIP("203.0.113.50", 3600);

// Permanent ban
sgcWebSocketFirewall1->BanIP("198.51.100.25", 0);

// Check and unban
if (sgcWebSocketFirewall1->IsBanned("203.0.113.50"))
  sgcWebSocketFirewall1->UnbanIP("203.0.113.50");
```

Configuration Example

```
// Strict configuration: 3 attempts in 30 seconds, ban for 10 minutes
sgcWebSocketFirewall1->BruteForce->Enabled = true;
sgcWebSocketFirewall1->BruteForce->MaxAttempts = 3;
sgcWebSocketFirewall1->BruteForce->TimeWindowSec = 30;
sgcWebSocketFirewall1->BruteForce->BanDurationSec = 600;
```

```
// Lenient configuration: 10 attempts in 5 minutes, ban for 1 minute
sgcWebSocketFirewall1->BruteForce->Enabled = true;
sgcWebSocketFirewall1->BruteForce->MaxAttempts = 10;
sgcWebSocketFirewall1->BruteForce->TimeWindowSec = 300;
sgcWebSocketFirewall1->BruteForce->BanDurationSec = 60;
```

Notes

- Whitelisted IPs are never banned, even if RegisterFailedAttempt is called for them.
- Bans are stored in memory by default. Use **SaveBansToFile** and **LoadBansFromFile** to persist bans across server restarts.
- The time window is sliding, not fixed. Each failed attempt is tracked individually and expires after TimeWindowSec seconds.

Firewall: SQL Injection and XSS Detection

TsgcWebSocketFirewall includes built-in pattern-based detection for SQL injection and Cross-Site Scripting (XSS) attacks in WebSocket messages. These modules scan incoming message content and take action when a threat pattern is detected.

SQL Injection Detection

The SQL injection module scans all incoming WebSocket messages for common SQL injection patterns. Detection is case-insensitive.

Properties

Property	Description
SQLInjection.Enabled	Enables or disables SQL injection detection. Default: False.
SQLInjection.Action	Action to take when a pattern is detected: **faDeny** (block the message and disconnect, default), **faLog** (fire OnViolation event but allow the message), **faAllow** (no action, detection disabled).
SQLInjection.CustomPatterns	TStringList of additional regex patterns to check. These are evaluated in addition to the built-in patterns.

Built-in Patterns

The following patterns are checked by default (case-insensitive):

Pattern	Attack Type
' OR ', ' AND '	Boolean-based injection
UNION SELECT	Union-based injection
'; DROP, '; DELETE, '; INSERT, '; UPDATE	Statement injection / data manipulation
-- (double dash)	SQL comment injection
EXEC(, EXECUTE(	Command execution
xp_cmdshell	Extended stored procedure abuse
CAST(, CONVERT(	Type conversion abuse
1=1, 1'='1	Tautology injection

Custom Patterns

You can add application-specific patterns using the CustomPatterns property:

```
sgcWebSocketFirewall1->SQLInjection->Enabled = true;
sgcWebSocketFirewall1->SQLInjection->CustomPatterns->Add("WAITFOR DELAY");
sgcWebSocketFirewall1->SQLInjection->CustomPatterns->Add("BENCHMARK\\(");
sgcWebSocketFirewall1->SQLInjection->CustomPatterns->Add("SLEEP\\(");
```

XSS Detection

The XSS module detects Cross-Site Scripting patterns in WebSocket messages, preventing injection of malicious scripts that could be rendered by other connected clients.

Properties

Property	Description
XSS.Enabled	Enables or disables XSS detection. Default: False.
XSS.Action	Action to take when a pattern is detected: **faDeny** (block and disconnect, default), **faLog** (fire OnViolation but allow), **faAllow** (no action).

Built-in Patterns

The following XSS patterns are detected (case-insensitive):

Pattern	Attack Type
<script	Script tag injection
javascript:	Protocol-based XSS
onerror=, onload=, onclick=, onmouseover=	Event handler injection
eval(	JavaScript code evaluation
document.cookie	Cookie theft
<iframe, <object, <embed	Element injection
<svg onload	SVG-based XSS
expression(	CSS expression injection

Action Property

Both SQLInjection and XSS modules support an Action property of type TsgcFirewallAction:

Value	Behavior
faDeny	The message is blocked and the client is disconnected. The OnViolation event fires. This is the default.
faLog	The message is allowed through, but the OnViolation event fires for logging purposes.

faAllow	No action is taken. Effectively disables detection while keeping the Enabled flag True.

OnViolation Event

When a SQL injection or XSS pattern is detected (with Action set to faDeny or faLog), the OnViolation event fires with details about the violation:

```
void __fastcall TForm1::FirewallViolation(TObject *Sender, const String aIP,
  const TsgcFirewallViolationType aViolationType, const String aDetails)
{
  switch (aViolationType)
  {
    case fvSQLInjection:
      LogWarning("SQL injection attempt from " + aIP + ": " + aDetails);
      break;
    case fvXSS:
      LogWarning("XSS attempt from " + aIP + ": " + aDetails);
      break;
  }
}
```

Configuration Example

```
// Enable SQL injection detection with deny action
sgcWebSocketFirewall1->SQLInjection->Enabled = true;
sgcWebSocketFirewall1->SQLInjection->Action = faDeny;

// Add custom patterns for time-based SQL injection
sgcWebSocketFirewall1->SQLInjection->CustomPatterns->Add("WAITFOR DELAY");
sgcWebSocketFirewall1->SQLInjection->CustomPatterns->Add("SLEEP\\(");

// Enable XSS detection in log-only mode (monitor without blocking)
sgcWebSocketFirewall1->XSS->Enabled = true;
sgcWebSocketFirewall1->XSS->Action = faLog;

// Assign violation handler
sgcWebSocketFirewall1->OnViolation = FirewallViolation;
```

Notes

- Pattern detection is applied to all incoming text messages. Binary messages are not scanned.
- Whitelisted IPs bypass message filtering entirely.
- The built-in patterns cover the most common attack vectors. For application-specific threats, add custom patterns via SQLInjection.CustomPatterns.
- Use faLog mode during initial deployment to monitor detections without impacting clients, then switch to faDeny once you have validated the patterns for your application.

Firewall: Rate Limiting and Flood Protection

TsgcWebSocketFirewall provides two complementary mechanisms for controlling traffic: connection rate limiting (controlling how many connections an IP can open) and message flood protection (controlling how many messages a connected client can send per second).

Connection Rate Limiting

Rate limiting restricts the number of connections a single IP address can establish within a time window. This prevents a single client from consuming all available server connections.

Properties

Property	Default	Description
RateLimit.Enabled	False	Enables or disables connection rate limiting.
RateLimit.MaxConnectionsPerIP	10	Maximum number of concurrent connections allowed from a single IP address.
RateLimit.TimeWindowSec	60	Time window in seconds for tracking connections. Connections older than this are removed from the count.

How It Works

1. When a new connection arrives, the firewall counts the number of active connections from that IP.
2. If the count is at or above MaxConnectionsPerIP, the new connection is rejected.
3. When a connection disconnects, it is removed from the tracking count.
4. Connections older than TimeWindowSec are automatically cleaned from tracking.

Example

```
// Allow maximum 5 connections per IP
sgcWebSocketFirewall1->RateLimit->Enabled = true;
sgcWebSocketFirewall1->RateLimit->MaxConnectionsPerIP = 5;
sgcWebSocketFirewall1->RateLimit->TimeWindowSec = 60;
```

Message Flood Protection

Flood protection limits the number of messages a single IP can send per second. This prevents abusive clients from overwhelming the server with rapid message bursts.

Properties

Property	Default	Description
FloodProtection.Enabled	False	Enables or disables message flood protection.

FloodProtection.MaxMessagesPerSec	100	Maximum number of messages allowed per second from a single IP address.
FloodProtection.Action	faDeny	Action when the limit is exceeded: **faDeny** (disconnect the client), **faLog** (fire OnViolation but allow), **faAllow** (no action).

How It Works

1. The firewall maintains a per-IP message counter that resets every second.
2. Each incoming message increments the counter for the sender's IP.
3. When the counter exceeds MaxMessagesPerSec, the configured Action is taken.
4. With faDeny action, the client is disconnected immediately. With faLog, the OnViolation event fires but the message is processed normally.

Example

```
// Limit to 50 messages per second per IP
sgcWebSocketFirewall1->FloodProtection->Enabled = true;
sgcWebSocketFirewall1->FloodProtection->MaxMessagesPerSec = 50;
sgcWebSocketFirewall1->FloodProtection->Action = faDeny;
```

Combining Both Protections

Rate limiting and flood protection work together to provide comprehensive traffic control:

```
// Connection limiting: max 5 connections per IP
sgcWebSocketFirewall1->RateLimit->Enabled = true;
sgcWebSocketFirewall1->RateLimit->MaxConnectionsPerIP = 5;

// Message limiting: max 50 messages/sec per IP
sgcWebSocketFirewall1->FloodProtection->Enabled = true;
sgcWebSocketFirewall1->FloodProtection->MaxMessagesPerSec = 50;

// Assign to server
sgcWebSocketHTTPServer1->Firewall = sgcWebSocketFirewall1;
```

With this configuration, each IP can open at most 5 concurrent connections, and across all those connections, it can send a combined maximum of 50 messages per second.

Tuning Guidelines

Scenario	MaxConnectionsPerIP	MaxMessagesPerSec
Chat application	3-5	10-30
Real-time data feed	2-3	100-500
Gaming server	1-2	50-100
API gateway	10-20	200-1000

Start with conservative limits and increase them based on monitoring. Use faLog mode initially to observe traffic patterns before enabling faDeny.

Notes

- Whitelisted IPs bypass both rate limiting and flood protection.
- Rate limit tracking data is cleared automatically when connections close. Use ClearTracking to reset all counters manually.
- Behind a reverse proxy, all connections arrive from the proxy address. Enable **ForwardedHeaders** so the firewall resolves the real client address from the X-Forwarded-For and X-Real-IP headers, see Firewall: Real Client IP Behind a Reverse Proxy.
- MaxConnectionsPerIP keeps counting the sockets opened by the proxy even with ForwardedHeaders enabled, because connection rate limiting is connection scoped and a proxy carries many end clients over a few upstream connections. Size the limit for the proxy, or whitelist its address. Flood protection is evaluated per message against the resolved address, so it does see the individual client.

Firewall: Bot Detection

TsgcWebSocketFirewall can classify connecting IP addresses as bots, crawlers, datacenter hosts or human visitors. Bot Detection combines a known-bot database, forward-confirmed reverse DNS verification, datacenter/ASN range matching and DNSBL blocklist lookups to assign each IP a classification. It is a **classify-only** module: it never blocks a connection by itself. It reports the verdict so your application can decide what to do.

How It Works

1. An IP connects to the server.
2. The firewall looks up the IP in the result cache. If a cached classification exists (and has not expired), it is used immediately.
3. On a cache miss, the IP is matched against the known-bots file (verified crawlers) and the datacenter ASN file (cloud/hosting ranges).
4. If ReverseDNS or DNSBL verification is enabled, a lookup is scheduled on a background thread so it never blocks the accept thread. Until the lookup completes, the IP may be classified as **bcUnknown**.
5. When a classification is resolved, it is stored in the cache (for CacheDurationSec seconds) and the **OnBotDetected** event fires.
6. The classification updates the firewall Stats (the **fvBot** violation counter) and is available to Custom Rules through the rule's **BotType** property.

Properties

Property	Description
BotDetection.Enabled	Enables or disables bot detection. Default: False.
BotDetection.KnownBotsFile	Path to a CSV file of verified crawlers. Each line is **CIDR,botname** (e.g. an IP range and the crawler it belongs to).
BotDetection.CacheDurationSec	Number of seconds a resolved classification is kept in the result cache before it is looked up again. Default: 3600.
BotDetection.DNSTimeoutMS	Timeout in milliseconds for each DNS lookup (reverse DNS and DNSBL). Default: 2000.
BotDetection.ReverseDNS.Enabled	Enables forward-confirmed reverse DNS (FCrDNS) verification of claimed crawlers. Default: False.
BotDetection.Datacenter.Enabled	Enables datacenter / cloud-hosting range detection. Default: False.
BotDetection.Datacenter.ASNFile	Path to a CSV file of datacenter ASN ranges. Each line is **CIDR,asnname** (e.g. a cloud provider range and its ASN name).
BotDetection.DNSBL.Enabled	Enables DNSBL (DNS blocklist) lookups. Default: False.
BotDetection.DNSBL.Zones	TStringList of DNSBL zones to query (for example **zen.spamhaus.org**).

Classification Values

The **TsgcBotClassification** enumeration describes the verdict assigned to an IP:

Value	Description
bcUnknown	The IP has not been classified yet. Returned on a cache miss while an asynchronous lookup is in progress.
bcVerified-Crawler	A known, legitimate crawler (for example a search engine bot) confirmed by the known-bots file or by forward-confirmed reverse DNS.
bcDatacenter	The IP belongs to a cloud or hosting provider range listed in the ASN file. Often automated traffic rather than a residential user.
bcSuspected-Bot	The IP shows bot characteristics but could not be matched to a verified crawler.
bcBlocklisted	The IP is listed on one of the configured DNSBL zones.
bcHuman	The IP looks like a regular human visitor (no bot, datacenter or blocklist match).

CSV File Formats

The known-bots file (**KnownBotsFile**) lists verified crawler ranges. Each line is a CIDR range followed by the bot name:

```
66.249.64.0/19,Googlebot
157.55.0.0/16,Bingbot
17.58.0.0/16,Applebot
```

The datacenter ASN file (**Datacenter.ASNFile**) lists cloud/hosting ranges. Each line is a CIDR range followed by the ASN name:

```
3.0.0.0/15,AMAZON-AES
13.64.0.0/11,MICROSOFT-AZURE
34.64.0.0/10,GOOGLE-CLOUD
```

Enabling Bot Detection

Enable the module, load the known-bot and ASN files, and turn on the verification sub-modules you need. DNS lookups run on a background thread, so enabling ReverseDNS or DNSBL does not slow down the accept thread.

```
// Enable bot detection
sgcWebSocketFirewall1->BotDetection->Enabled = true;
sgcWebSocketFirewall1->BotDetection->KnownBotsFile = "C:\\data\\known_bots.csv";
sgcWebSocketFirewall1->BotDetection->CacheDurationSec = 3600;
sgcWebSocketFirewall1->BotDetection->DNSTimeoutMS = 2000;

// Forward-confirmed reverse DNS verification
sgcWebSocketFirewall1->BotDetection->ReverseDNS->Enabled = true;

// Datacenter / cloud-hosting detection
sgcWebSocketFirewall1->BotDetection->Datacenter->Enabled = true;
sgcWebSocketFirewall1->BotDetection->Datacenter->ASNFile = "C:\\data\\datacenter_asn.csv";

// DNSBL blocklist lookups
sgcWebSocketFirewall1->BotDetection->DNSBL->Enabled = true;
sgcWebSocketFirewall1->BotDetection->DNSBL->Zones->Add("zen.spamhaus.org");
```

Classify-Only Behavior and OnBotDetected

Bot Detection never rejects a connection on its own. When an IP is classified it fires the **OnBotDetected** event, updates the firewall Stats (the **fvBot** counter) and stores the result in the cache. Use the event to log traffic, build analytics, or feed your own logic:

```
void __fastcall TForm1::FirewallBotDetected(TObject *Sender, const String aIP,
  TsgcBotClassification aClassification, const String aBotName)
{
  switch (aClassification)
  {
    case bcVerifiedCrawler: Log("Verified crawler " + aBotName + " from " + aIP); break;
    case bcDatacenter:      Log("Datacenter host from " + aIP); break;
    case bcBlocklisted:     Log("Blocklisted IP " + aIP); break;
  }
}
```

Acting on a Classification with a Custom Rule

Because Bot Detection only classifies, blocking or banning is done through the Custom Rules engine. Each **TsgcFirewallRuleItem** exposes a **BotType** property: when set, the rule matches connections whose classification equals that value, and you can then deny, ban or log them.

```
sgcWebSocketFirewall1->CustomRules->Enabled = true;

// Deny connections coming from blocklisted IPs
TsgcFirewallRuleItem *vRule =
  (TsgcFirewallRuleItem *)sgcWebSocketFirewall1->CustomRules->Rules->Add();
vRule->BotType = bcBlocklisted;
vRule->ActionType = raDeny;
```

Custom Verdicts with OnResolveBot

If you have your own bot intelligence (a third-party reputation service, an internal allow/deny list, etc.), assign the **OnResolveBot** event. When assigned, your handler supplies the verdict instead of the built-in resolver. Set the *aClassification* and *aBotName* var parameters to return your result:

```
void __fastcall TForm1::FirewallResolveBot(TObject *Sender, const String aIP,
  TsgcBotClassification &aClassification, String &aBotName)
{
  // Custom lookup - query your own bot intelligence service
  if (MyBotService->IsCrawler(aIP))
  {
    aClassification = bcVerifiedCrawler;
    aBotName = MyBotService->GetName(aIP);
  }
  else
    aClassification = bcHuman;
}
```

Note: When OnResolveBot is assigned, the built-in resolver (known-bots file, reverse DNS, datacenter and DNSBL checks) is bypassed for that IP. The verdict your handler returns is cached and passed on to OnBotDetected and Custom Rules exactly like a built-in result.

Firewall: Real Client IP Behind a Reverse Proxy

When the server runs behind a reverse proxy such as Caddy, NGINX or HAProxy, every connection reaches the server from the proxy address. Blacklists, bans, GeoIP filtering, custom rules and audit entries would all see the proxy instead of the end client. The ForwardedHeaders property tells TsgcWebSocketFirewall to take the real client address from the X-Forwarded-For and X-Real-IP request headers, so every later decision is made against the client that actually sent the request.
The feature is disabled by default. Nothing changes when you upgrade until you set ForwardedHeaders.Enabled to True.

Properties

Property	Default	Description
ForwardedHeaders.Enabled	False	Enables or disables reverse-proxy address resolution. While False the socket address is always used and no header is read.
ForwardedHeaders.TrustedProxies	(empty)	TStringList with the addresses of the proxies that are allowed to supply forwarded headers. Accepts plain IP addresses and CIDR ranges, the same syntax used by Whitelist.IPs and Blacklist.IPs. While the list is empty the headers are ignored.
ForwardedHeaders.Mode	fwmBoth	Which headers are read: **fwmXForwardedFor** (only X-Forwarded-For), **fwmXRealIP** (only X-Real-IP), **fwmBoth** (X-Real-IP first, and the X-Forwarded-For chain when X-Real-IP is absent or empty).
ForwardedHeaders.TrustedHops	1	Number of proxies in front of the server. Each one appends an entry, so the last TrustedHops entries of the X-Forwarded-For chain come from the proxies and the client address is the first of them. A shorter chain is not trusted and the socket address is kept.

The Trust Gate

Forwarded headers travel inside the request, so anyone who can reach the server can write any value into them. The firewall therefore honours them **only** when the immediate peer, the address of the socket that opened the connection, matches an entry in TrustedProxies. A request that arrives directly from the internet keeps its socket address, because the internet peer is not in the list.
With an empty TrustedProxies list the headers are ignored entirely. This is deliberate and it is the anti-spoofing gate of the whole feature. Without a trust anchor any client could claim any address, write false entries into your audit trail, and step around a blacklist or an active ban simply by rotating a header value.
List every address the proxy may connect from. When the proxy runs on the same machine that is usually 127.0.0.1, and a container or load-balancer subnet can be given in CIDR form, for example 10.0.0.0/8.

Reading the X-Forwarded-For Chain

X-Forwarded-For is a comma separated list that grows from left to right as the request passes through each hop. The leftmost entry is the one the client itself supplied, so it is attacker controlled and must never be taken at face value. Each proxy appends the address of the peer it received the request from, so the last TrustedHops entries of the chain are the ones the trusted proxies wrote, and the firewall takes the first of them as the client address. Everything to the left of that entry came from the client, so it is never used. With a single appending proxy and TrustedHops set to 1, a chain of 1.2.3.4, 203.0.113.7 resolves to 203.0.113.7, and the 1.2.3.4 the client wrote itself is ignored. When the chain holds fewer entries than TrustedHops it does not match the topology you configured, nothing in it can be trusted and the socket address is kept. A TrustedHops of 0 means no trusted hop appended anything, so the rightmost entry is used.
Set TrustedHops to the number of proxies in front of the server. The default value of 1 covers the common case of a single reverse proxy. Increase it when a CDN or an outer load balancer forwards to your own proxy.
The safest arrangement is the one shown in the examples below, where the proxy **sets** X-Forwarded-For to the address it received the request from instead of appending to whatever the client sent. The chain then holds a single trustworthy entry and there is no client-supplied prefix at all.

How It Works

1. The connection is accepted and the connect-time check runs against the socket peer, which is the proxy. This happens before the TLS handshake, so the proxy address itself must be accepted by Whitelist and Blacklist.
2. A request arrives. If ForwardedHeaders.Enabled is False the socket address is used and nothing else happens.
3. The peer address is compared against TrustedProxies. If it does not match, or the list is empty, the headers are ignored and the socket address is kept.
4. A candidate address is taken from the headers selected by Mode. Brackets, an IPv6 zone identifier and a trailing port are stripped, and the value is normalized.
5. The candidate is parsed as IPv4 or IPv6. Anything that does not parse is discarded and the peer address is kept, because an unparseable value would not match any rule.
6. The connection IP property is overwritten with the resolved address. Every later firewall check, every event and your own handler code now see the end client. The original socket address stays available in the connection PeerIP property.
7. The resolved address is passed to IsForwardedIPAllowed, which applies the address-identity checks (whitelist, blacklist, bans, GeoIP, custom rules). A rejected request is answered with 403 Forbidden by the HTTP servers, and the connection is dropped for WebSocket and the other protocol handlers.
Resolution runs **per request** and not per connection, because a reverse proxy reuses a single upstream connection for requests coming from different end clients. Each request resolves against the frozen socket address, never against the address the previous request left behind.

Caddy Example

A Caddyfile that terminates TLS on the public interface and forwards to the server listening on port 8080:

```
example.com {
    reverse_proxy http://127.0.0.1:8080 {
        header_up X-Forwarded-For {remote_host}
        header_up X-Forwarded-Host {host}
        header_up X-Forwarded-Proto {scheme}
        header_up X-Real-IP {remote_host}
    }
}
```

The matching Delphi configuration. Caddy connects from the loopback address, so 127.0.0.1 is both the trusted proxy and a whitelisted address:

```
// Resolve the real client address from the proxy headers
sgcWebSocketFirewall1->ForwardedHeaders->Enabled = true;
sgcWebSocketFirewall1->ForwardedHeaders->TrustedProxies->Add("127.0.0.1");
sgcWebSocketFirewall1->ForwardedHeaders->Mode = fwmBoth;
sgcWebSocketFirewall1->ForwardedHeaders->TrustedHops = 1;

// The connect-time check sees the proxy, so the proxy must be allowed
sgcWebSocketFirewall1->Whitelist->Enabled = true;
```

```
sgcWebSocketFirewall1->Whitelist->IPs->Add("127.0.0.1");

sgcWebSocketFirewall1->Enabled = true;
sgcWebSocketHTTPServer1->Firewall = sgcWebSocketFirewall1;
```

The whitelist is optional. If you do not enable it, make sure instead that the proxy address is neither blacklisted nor banned. What must never happen is the proxy address being rejected by the connect-time check.

NGINX Example

The equivalent NGINX location block, including the two headers a WebSocket upgrade needs:

```
location / {
    proxy_pass http://127.0.0.1:8080;
    proxy_set_header X-Forwarded-For $remote_addr;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_set_header Host $host;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}
```

The Delphi side is identical to the Caddy example. Only the trusted proxy address changes when NGINX runs on another host.

Resolved Address and Socket Address

Property	Value
Connection.IP	The real client address once resolution succeeds. This is what every firewall check, every event and your own code read. Without resolution it stays the socket address.
Connection.PeerIP	Always the original socket address, which is the proxy. It is captured the first time the address is replaced and never changes afterwards.

```
void __fastcall TForm1::OnMessage(TsgcWSConnection *Connection, const UnicodeString Text)
{
  // Connection->IP is the end client, Connection->PeerIP is the proxy
  Memo1->Lines->Add(Connection->IP + " via " + Connection->PeerIP + ": " + Text);
}
```

Rate Limiting Stays Connection Scoped

RateLimit.MaxConnectionsPerIP and the internal connection counters track sockets, and a reverse proxy opens its own sockets. They keep counting the proxy connections even with ForwardedHeaders enabled, and they are not re-evaluated per request. IsForwardedIPAllowed is the per-request verdict and it deliberately leaves the rate limit out, so a proxy that multiplexes many end clients over a few upstream connections is never rejected for exceeding a per-IP connection cap.
Set MaxConnectionsPerIP high enough for the number of upstream connections your proxy opens, or whitelist the proxy address so the limit is bypassed altogether. To limit individual clients behind the proxy, use FloodProtection, which is evaluated per message against the resolved address, or a custom rule.

Troubleshooting

Everything is blocked once the proxy is in front of the server

This is by far the most common misconfiguration. The connect-time check runs on the socket peer before TLS and before any header is read, so it sees the proxy. If the proxy address is not accepted at that point, the proxy itself is refused and nothing else in the chain ever runs. Add the proxy address to Whitelist.IPs, or make sure it is neither blacklisted nor banned. Whitelist.Enabled with a list that does not contain the proxy rejects everything.

The headers are ignored and every client still shows the proxy address

Check TrustedProxies. An empty list disables header reading entirely, and an entry that does not match the address the proxy actually connects from has the same effect. Compare Connection.PeerIP against what you configured. A proxy on the same machine may connect from 127.0.0.1 or from ::1, and both may need to be listed. Also confirm that ForwardedHeaders.Enabled is True and that the proxy really sends the header, since Mode fwmXRealIP ignores X-Forwarded-For and fwmXForwardedFor ignores X-Real-IP.

The resolved address is another proxy instead of the client

TrustedHops is too low for the number of hops in front of the server. The entry that gets taken sits too far to the right in the chain, so the address one of the inner proxies appended for its own peer is resolved instead of the client. Increase TrustedHops by one for each extra proxy, or configure the outermost proxy under your control to set X-Forwarded-For rather than append to it.

The address falls back to the proxy even though TrustedProxies matches

TrustedHops is higher than the number of entries the chain actually carries. The chain does not match the configured topology, so resolution fails closed and the socket address stays in place, which is why the audit shows the proxy. Seeing the proxy address here is the signal that the setting is wrong, not that the headers are missing. Count the hops that really append an entry, remembering that a proxy configured to set X-Forwarded-For instead of appending leaves a chain of one, and lower TrustedHops to match.

Notes

- Both IPv4 and IPv6 addresses are resolved. A value such as [2001:db8::1]:443 is accepted, and the brackets, the zone identifier and the port are removed before the address is used.
- A malformed header never takes the server down. The request is let through with the peer address untouched.
- ResolveClientIP is public, so you can resolve an address yourself from a TStrings of headers or from two explicit header values without going through the server.
- The feature covers the HTTP servers and the WebSocket and protocol handlers built on the Indy server core.

TsgcWSRateLimiter

TsgcWSRateLimiter implements a full-featured rate limiting component that protects server endpoints from excessive traffic, abuse and scraping.

Introduction

TsgcWSRateLimiter implements a full-featured rate limiting component that protects server endpoints from excessive traffic, abuse and scraping. It supports three algorithms (Token Bucket, Sliding Window, Fixed Window), scoped rules (per IP, per API key, per user, per endpoint), long-term quotas (hour / day / month), burst protection, a configurable HTTP 429 response shape and real-time statistics. Internal state can be persisted to a file or stream so counters survive restarts.

The rate limiter integrates automatically with server components through the **RateLimiter** property available on TsgcWebSocketHTTPServer and TsgcWSServer_HTTPAPI. Once assigned, it evaluates every connection and every message without requiring manual event wiring.

Quick Start

1. Drop a **TsgcWSRateLimiter** component on the form.

2. Enable one or more strategies and scopes (see the per-section documentation below).

3. Assign the rate limiter to a server component:

```
sgcWebSocketHTTPServer1.RateLimiter := sgcWSRateLimiter1;
```

4. Start the server. The rate limiter automatically evaluates every connection and message, rejects excess traffic with HTTP 429 and updates its statistics counters.

Top-level Properties

What it does: Master switch for the component and optional storage file used by the state-persistence methods. When Enabled is False every public method short-circuits to "allowed" and no counters are updated.

When to use it: Disable the rate limiter at runtime during a controlled load test or during a maintenance window without unassigning it from the server. Set StorageFile so counters survive an application restart — for example a rolling Token Bucket continues refilling from where it stopped instead of starting fresh and accepting a huge burst.

Property	Type	Default	Description
Enabled	Boolean	True	Master switch. When False, IsAllowed / Consume / IsConnectionAllowed / IsMessageAllowed always return True and no counters are updated.
StorageFile	string	''	Optional file path used by SaveStateToFile / LoadStateFromFile for counter persistence across restarts.

```
sgcWSRateLimiter1->Enabled = true;
sgcWSRateLimiter1->StorageFile = "ratelimit.dat";
sgcWSRateLimiter1->LoadStateFromFile(sgcWSRateLimiter1->StorageFile);
```

TokenBucket Strategy

What it does: Maintains a bucket of up to *Capacity* tokens per key. The bucket refills at *RefillRate* tokens every *RefillIntervalMs* milliseconds. Each request consumes one token (or the Cost argument passed to Consume). When the bucket is empty the request is rejected and a RetryAfterSec hint is computed from the refill schedule.

When to use it: The API advertises an average rate of 10 req/sec but you want to let well-behaved clients burst up to 50 requests back-to-back without being throttled. A Token Bucket with Capacity=50 and RefillRate=10 per 1000 ms lets a client send 50 requests instantly, then forces them down to 10 req/sec until the bucket refills.

Property	Type	Default	Description
Enabled	Boolean	False	Enables the Token Bucket algorithm as the global default strategy.
Capacity	Integer	100	Maximum number of tokens the bucket can hold. Controls the maximum burst size.
RefillRate	Integer	10	Number of tokens added to the bucket every RefillIntervalMs.
RefillIntervalMs	Integer	1000	Refill period in milliseconds. Together with RefillRate determines the sustained request rate.

```
// Average 10 req/sec, bursts up to 50
sgcWSRateLimiter1->TokenBucket->Enabled = true;
sgcWSRateLimiter1->TokenBucket->Capacity = 50;
sgcWSRateLimiter1->TokenBucket->RefillRate = 10;
sgcWSRateLimiter1->TokenBucket->RefillIntervalMs = 1000;
// First 50 requests succeed immediately; afterwards throttled to 10/sec until refill
```

SlidingWindow Strategy

What it does: Stores a timestamp for every request and counts the timestamps that fall within a rolling window of *WindowSec* seconds. Rejects when the count would exceed *MaxRequests*. The window slides continuously, so there is no hard reset moment where a burst can slip through.

When to use it: You publish a strict SLA guaranteeing "no more than 100 requests in any 60-second rolling window". Fixed Window would let a client fire 100 requests at 12:00:59 and another 100 at 12:01:00 (200 within one second). Sliding Window prevents that and enforces the SLA exactly.

Property	Type	Default	Description

Enabled	Boolean	False	Enables the Sliding Window algorithm as the global default strategy.
WindowSec	Integer	60	Size of the rolling window in seconds.
MaxRequests	Integer	1000	Maximum number of requests permitted within any WindowSec-wide rolling window.

```
// Strict SLA: max 100 requests in any rolling 60 seconds
sgcWSRateLimiter1->SlidingWindow->Enabled = true;
sgcWSRateLimiter1->SlidingWindow->WindowSec = 60;
sgcWSRateLimiter1->SlidingWindow->MaxRequests = 100;
// Client that sends a burst of 100 requests near t=59s will be throttled at t=60s
```

FixedWindow Strategy

What it does: Counts requests within a fixed clock window of *WindowSec* seconds and resets the counter to zero at the window boundary. The simplest algorithm and the cheapest memory-wise; however it can allow up to 2×MaxRequests in a short period around a boundary crossing.

When to use it: Billing-style quotas like "1000 requests per hour, counter resets on the hour". Customers expect a clearly visible reset time, and the minor boundary-burst behavior is acceptable because the billing unit is the hour, not the rolling minute.

Property	Type	Default	Description
Enabled	Boolean	False	Enables the Fixed Window algorithm as the global default strategy.
WindowSec	Integer	60	Size of the fixed window in seconds. The counter resets to 0 when the window elapses.
MaxRequests	Integer	1000	Maximum number of requests permitted within a single fixed window.

```
// Billing-style: 1000 requests per hour, resets on the hour
sgcWSRateLimiter1->FixedWindow->Enabled = true;
sgcWSRateLimiter1->FixedWindow->WindowSec = 3600;
sgcWSRateLimiter1->FixedWindow->MaxRequests = 1000;
```

PerIP

What it does: Defines a rate limit scoped to the source IP address. When a request key does not match PerAPIKey / PerUser / PerEndpoint, PerIP becomes the active rule and overrides the global Token/Sliding/Fixed defaults. Each IP gets its own independent counter.

When to use it: A public API with many clients. Without PerIP a single noisy client can consume the global budget and starve everyone else. Give every IP its own bucket (for example 100 requests per minute) so one client cannot affect others.

Property	Type	Default	Description
Enabled	Boolean	False	Enables per-IP rate limiting.
MaxRequests	Integer	100	Maximum requests per window (or bucket capacity when Strategy is rlsTokenBucket).
WindowSec	Integer	60	Window size in seconds. For Token Bucket also used to derive the refill rate (MaxRequests / WindowSec tokens per second).
Strategy	TsgcRateLimitStrategy	rlsTokenBucket	Algorithm used for this scope: rlsTokenBucket, rlsSliding or rlsFixed.

```
// Every client IP gets its own 100 req/min bucket
sgcWSRateLimiter1->PerIP->Enabled = true;
sgcWSRateLimiter1->PerIP->Strategy = rlsTokenBucket;
sgcWSRateLimiter1->PerIP->MaxRequests = 100;
sgcWSRateLimiter1->PerIP->WindowSec = 60;
```

PerAPIKey

What it does: Same semantics as PerIP but scoped to an API-key identifier. The component recognizes keys that start with the prefix `apikey:` as API-key-scoped. Designed to integrate with TsgcWSAPIKeyManager so every customer gets an independent rate budget regardless of source IP.

When to use it: SaaS with tiered pricing. Every customer has their own API key and their own quota independent of where they connect from. Use PerAPIKey as the default tier and differentiate customer tiers (Free / Pro / Enterprise) either through a PerEndpoint rule keyed by path prefix or by overriding the decision inside the OnThrottled event handler.

Property	Type	Default	Description
Enabled	Boolean	False	Enables per-API-key rate limiting.
MaxRequests	Integer	100	Maximum requests per window (or bucket capacity for Token Bucket).
WindowSec	Integer	60	Window size in seconds.

Strategy	TsgcRate-LimitStrat-egy	rlsToken-Bucket	Algorithm used for this scope.

```
// Default API-key tier: 1000 req/min
sgcWSRateLimiter1->PerAPIKey->Enabled = true;
sgcWSRateLimiter1->PerAPIKey->Strategy = rlsSliding;
sgcWSRateLimiter1->PerAPIKey->MaxRequests = 1000;
sgcWSRateLimiter1->PerAPIKey->WindowSec = 60;
// Tier-differentiation can be done via PerEndpoint rules or OnThrottled
```

PerUser

What it does: Same structure as PerIP / PerAPIKey but keyed by an application-defined user identifier. The component recognizes keys that start with the prefix `user:` as user-scoped. Useful when a single user may connect from multiple devices or IPs and you want a single shared budget for that user.

When to use it: Authenticated web / mobile app. User `alice@example.com` is allowed 1000 requests per day regardless of whether she calls from her phone, laptop or work desktop. Aggregating by IP would double-count; aggregating by user gives the budget you actually want to bill against.

Property	Type	Default	Description
Enabled	Boolean	False	Enables per-user rate limiting.
MaxRe-quests	Integer	100	Maximum requests per window (or bucket capacity for To-ken Bucket).
WindowSec	Integer	60	Window size in seconds.
Strategy	TsgcRate-LimitStrat-egy	rlsToken-Bucket	Algorithm used for this scope.

```
// 1000 requests/day per authenticated user across all their devices
sgcWSRateLimiter1->PerUser->Enabled = true;
sgcWSRateLimiter1->PerUser->Strategy = rlsFixed;
sgcWSRateLimiter1->PerUser->MaxRequests = 1000;
sgcWSRateLimiter1->PerUser->WindowSec = 86400;
// Pass keys like "user:alice@example.com" to Consume / IsAllowed
```

PerEndpoint

What it does: Collection of pattern-based rules. Each item has its own *Pattern* (wildcards `*` and `?` supported), *Strategy*, *MaxRequests* and *WindowSec*. The first enabled item whose pattern matches the key wins — no further items are evaluated. Takes precedence over PerAPIKey, PerUser and PerIP.

When to use it: You have an expensive endpoint `/api/v1/expensive-report` that must be limited to 10 req/min per client, while a cheap status endpoint `/api/v1/status` allows 1000 req/min. Different endpoints have different budgets — configure one PerEndpoint item per path pattern.

Property	Type	Default	Description
Enabled	Boolean	False	Enables the PerEndpoint rule set. When False no items are evaluated.
Rules	TCollection	empty	Collection of TsgcRateLimitRuleItem. Order matters — first match wins.

Each **TsgcRateLimitRuleItem** exposes:

Property	Type	Default	Description
Name	string	''	Friendly name shown in the object inspector.
Enabled	Boolean	True	Disables this item without deleting it.
Pattern	string	''	Wildcard pattern matched against the key (case-insensitive). Supports `*` and `?`.
MaxRequests	Integer	100	Maximum requests per window / bucket capacity.
WindowSec	Integer	60	Window size in seconds.
Strategy	TsgcRateLimitStrategy	rlsTokenBucket	Algorithm to use for matched requests.

```
sgcWSRateLimiter1->PerEndpoint->Enabled = true;

TsgcRateLimitRuleItem *ruleReport = (TsgcRateLimitRuleItem*)
  sgcWSRateLimiter1->PerEndpoint->Rules->Add();
ruleReport->Name = "expensive-report";
ruleReport->Pattern = "*/api/v1/expensive-report*";
ruleReport->Strategy = rlsSliding;
ruleReport->MaxRequests = 10;
ruleReport->WindowSec = 60;

TsgcRateLimitRuleItem *ruleStatus = (TsgcRateLimitRuleItem*)
  sgcWSRateLimiter1->PerEndpoint->Rules->Add();
ruleStatus->Name = "status";
ruleStatus->Pattern = "*/api/v1/status*";
ruleStatus->Strategy = rlsTokenBucket;
ruleStatus->MaxRequests = 1000;
ruleStatus->WindowSec = 60;
```

Quotas

What it does: Long-term caps that persist beyond the rolling / refilling window of the main strategy. Each item caps how many requests a key (or the whole system) may make within a calendar period (*qpHour*, *qpDay* or *qpMonth*). Once the *Limit* is reached the request is rejected and the OnQuotaExceeded event fires.

When to use it: Free-tier plan limited to 10,000 requests per month hard cap. The rolling token bucket still smooths short-term traffic, but when the monthly quota is exhausted every further call fails until the next calendar month.

Property	Type	Default	Description
Enabled	Boolean	False	Enables the quota rule set. When False no items are evaluated.
Quotas	TCollection	empty	Collection of TsgcRateLimitQuotaItem. All enabled items are evaluated in order.

Each **TsgcRateLimitQuotaItem** exposes:

Property	Type	Default	Description
Name	string	''	Friendly name reported in OnQuotaExceeded and in Result.Reason ("quota:Name").
Enabled	Boolean	True	Disables this quota without deleting it.
Scope	TsgcRateLimitQuotaScope	qscPerKey	qscPerKey counts independently per key; qscGlobal counts across all keys combined.
Period	TsgcRateLimitQuotaPeriod	qpHour	qpHour, qpDay or qpMonth. Counter resets automatically at the start of each period.
Limit	Int64	10000	Maximum requests permitted within the period.

```
sgcWSRateLimiter1->Quotas->Enabled = true;

TsgcRateLimitQuotaItem *quota = (TsgcRateLimitQuotaItem*)
  sgcWSRateLimiter1->Quotas->Quotas->Add();
quota->Name = "free-tier-monthly";
quota->Scope = qscPerKey;
quota->Period = qpMonth;
quota->Limit = 10000;
```

BurstProtection

What it does: A short-timescale spike detector that runs in parallel with the main strategy. If more than *BurstThreshold* requests arrive from the same key within *BurstWindowMs* milliseconds, the key is placed in a cooldown

for *CooldownSec* seconds during which every request is rejected — even if the main strategy would have allowed them.

When to use it: Detect scraper and scanner clients that flood a server with 100+ requests in 100 ms bursts but stay under a per-minute limit. Burst protection catches this machine-gun pattern that a human user never produces.

Property	Type	Default	Description
Enabled	Boolean	False	Enables the burst detector.
BurstThreshold	Integer	50	Number of requests within BurstWindowMs that triggers a cooldown.
BurstWindowMs	Integer	500	Burst observation window in milliseconds.
CooldownSec	Integer	30	Duration of the cooldown period after a burst is detected. During cooldown every request is rejected.

```
// More than 50 requests in 500ms triggers a 30-second cooldown
sgcWSRateLimiter1->BurstProtection->Enabled = true;
sgcWSRateLimiter1->BurstProtection->BurstThreshold = 50;
sgcWSRateLimiter1->BurstProtection->BurstWindowMs = 500;
sgcWSRateLimiter1->BurstProtection->CooldownSec = 30;
```

Response

What it does: Configures the shape of the HTTP response returned to clients when a request is throttled. Controls the status code, the body message and whether `Retry-After` and `X-RateLimit-*` headers are added to the response.

When to use it: You want well-behaved clients to back off correctly. Return the standard 429 status with a `Retry-After` header indicating how long they should wait, and include `X-RateLimit-Limit` / `X-RateLimit-Remaining` / `X-RateLimit-Reset` so clients can pace themselves proactively instead of reacting to rejections.

Property	Type	Default	Description
StatusCode	Integer	429	HTTP status code returned for throttled requests. Use 429 (standard) or 503 for maintenance-style rejection.
Message	string	"Too Many Requests"	Body of the HTTP response for throttled requests.
RetryAfterHeader	Boolean	True	When True adds a `Retry-After` header with the computed retry delay in seconds.
IncludeRateLimitHeaders	Boolean	True	When True adds `X-RateLimit-Limit` / `X-RateLimit-Remaining` / `X-RateLimit-Reset` headers to every response.

```
sgcWSRateLimiter1->Response->StatusCode = 429;
sgcWSRateLimiter1->Response->Message = "Too Many Requests — please slow down";
sgcWSRateLimiter1->Response->RetryAfterHeader = true;
sgcWSRateLimiter1->Response->IncludeRateLimitHeaders = true;
```

Stats

What it does: Read-only counter object that exposes live statistics about the rate limiter. All counters are updated atomically inside the component's internal critical section.

When to use it: Live dashboard in an admin UI. Poll Stats from a TTimer and paint the current TotalRequests / TotalThrottled / ActiveKeys in labels, gauges or charts so operators can see traffic pressure in real time.

Property	Type	Description
TotalRequests	Int64	Total number of requests evaluated since the last Reset.
TotalThrottled	Int64	Total number of requests rejected by the strategy or burst protection.
TotalQuotaExceeded	Int64	Total number of requests rejected because a quota was reached.
ActiveKeys	Integer	Number of keys currently tracked (buckets + window counters + active connections).
Uptime	Int64	Seconds since the stats object was last reset.

```
// Poll from a TTimer and update a dashboard
void __fastcall TForm1::StatsTimer(TObject *Sender)
{
  LabelRequests->Caption = "Requests: " + IntToStr(sgcWSRateLimiter1->Stats->TotalRequests);
  LabelThrottled->Caption = "Throttled: " + IntToStr(sgcWSRateLimiter1->Stats->TotalThrottled);
  LabelActiveKeys->Caption = "Active keys: " + IntToStr(sgcWSRateLimiter1->Stats->ActiveKeys);
  LabelUptime->Caption = "Uptime: " + IntToStr(sgcWSRateLimiter1->Stats->Uptime) + " s";
}
```

Events

OnThrottled: Fired when a request is rejected by the strategy or burst protection. Provides the key (IP, API key or user) and the reason string ("token_bucket", "sliding_window", "fixed_window", "burst", "connection_rate" or "message_rate"). The *Allow* parameter (var Boolean) lets you override the decision and let the request through anyway.

OnQuotaExceeded: Fired when a long-term quota is reached. Provides the key and the Name of the quota that was exceeded.

OnStateChange: Fired when the Token Bucket size changes for a key. Provides the key together with the old and new bucket sizes. Useful for telemetry, debugging and real-time visualizations.

Public Methods

Method	Description
IsAllowed(Key, Cost)	Returns True if a request of the given Cost (default 1) is allowed for the specified key.
Consume(Key, Cost)	Attempts to consume Cost tokens and returns a TsgcRateLimitResult (Allowed, Remaining, RetryAfterSec, Reason).
Reset(Key)	Clears the counters for the specified key.
ResetAll	Clears all internal counters for every tracked key and resets the Stats object.
GetRemaining(Key)	Returns the number of requests still available for the specified key.
GetRetryAfter(Key)	Returns the number of seconds the caller should wait before retrying.
SaveStateToFile(FileName)	Persists all buckets and counters to the given file.
LoadStateFromFile(FileName)	Restores buckets and counters previously saved to a file.
SaveStateToStream(Stream)	Persists the internal state to an arbitrary stream.
LoadStateFromStream(Stream)	Restores the internal state from an arbitrary stream.
IsConnectionAllowed(IP)	Server-integration hook. Returns True if a new connection from IP passes rate limiting. Called automatically by the server.
IsMessageAllowed(IP, Message)	Server-integration hook. Returns True if a message from IP passes rate limiting. Called automatically by the server.
RegisterConnection(IP)	Registers a new connection for tracking. Called automatically.
UnregisterConnection(IP)	Removes a connection from tracking. Called automatically on disconnect.

Automatic Integration

When assigned to a server's RateLimiter property, the component automatically:

- **On new connection:** Calls IsConnectionAllowed to evaluate PerIP, burst protection and the global strategy. Rejects the connection with HTTP 429 before the OnConnect event if the limit is exceeded. Calls RegisterConnection to track the client.
- **On message received:** Calls IsMessageAllowed to check PerIP, PerEndpoint, PerAPIKey, PerUser, quotas and burst protection. Disconnects or rejects the message when the limit is exceeded.
- **On disconnect:** Calls UnregisterConnection to release tracking resources.

Complete Example

```
// Enable rate limiter
sgcWSRateLimiter1->Enabled = true;

// Global Token Bucket: 50 burst, 10/sec sustained
sgcWSRateLimiter1->TokenBucket->Enabled = true;
sgcWSRateLimiter1->TokenBucket->Capacity = 50;
sgcWSRateLimiter1->TokenBucket->RefillRate = 10;
sgcWSRateLimiter1->TokenBucket->RefillIntervalMs = 1000;

// Per-IP sliding window: 60 requests per minute per IP
sgcWSRateLimiter1->PerIP->Enabled = true;
sgcWSRateLimiter1->PerIP->Strategy = rlsSliding;
sgcWSRateLimiter1->PerIP->MaxRequests = 60;
sgcWSRateLimiter1->PerIP->WindowSec = 60;

// Per-endpoint: expensive report endpoint = 10 req/min
sgcWSRateLimiter1->PerEndpoint->Enabled = true;
TsgcRateLimitRuleItem *rule = (TsgcRateLimitRuleItem*)
  sgcWSRateLimiter1->PerEndpoint->Rules->Add();
rule->Name = "expensive-report";
rule->Pattern = "*/api/v1/expensive-report*";
rule->Strategy = rlsSliding;
rule->MaxRequests = 10;
rule->WindowSec = 60;

// Burst protection: 50 requests in 500ms triggers 30s cooldown
sgcWSRateLimiter1->BurstProtection->Enabled = true;
sgcWSRateLimiter1->BurstProtection->BurstThreshold = 50;
sgcWSRateLimiter1->BurstProtection->BurstWindowMs = 500;
sgcWSRateLimiter1->BurstProtection->CooldownSec = 30;

// Monthly quota per API key: 10000 requests/month
sgcWSRateLimiter1->Quotas->Enabled = true;
TsgcRateLimitQuotaItem *quota = (TsgcRateLimitQuotaItem*)
  sgcWSRateLimiter1->Quotas->Quotas->Add();
quota->Name = "free-tier-monthly";
quota->Scope = qscPerKey;
quota->Period = qpMonth;
quota->Limit = 10000;

// HTTP 429 response with Retry-After and X-RateLimit-* headers
sgcWSRateLimiter1->Response->StatusCode = 429;
sgcWSRateLimiter1->Response->Message = "Too Many Requests";
sgcWSRateLimiter1->Response->RetryAfterHeader = true;
sgcWSRateLimiter1->Response->IncludeRateLimitHeaders = true;

// Persist state across restarts
sgcWSRateLimiter1->StorageFile = "ratelimit.dat";
sgcWSRateLimiter1->LoadStateFromFile("ratelimit.dat");

// Assign to server
sgcWebSocketHTTPServer1->RateLimiter = sgcWSRateLimiter1;

// Handle throttled events
void __fastcall TForm1::RateLimiterThrottled(TObject *Sender,
  const String aKey, const String aReason, bool &Allow)
{
  Log("Throttled " + aKey + ": " + aReason);
}

// Handle quota exhausted events
void __fastcall TForm1::RateLimiterQuotaExceeded(TObject *Sender,
  const String aKey, const String aQuotaName)
{
  Log("Quota " + aQuotaName + " exceeded for " + aKey);
}
```

Thread Safety

The rate limiter is fully thread-safe. All public methods use internal critical sections and thread-safe counters to protect concurrent access. The component can be shared by multiple server instances and accessed from any thread (server event handlers, timer threads, background workers) without external synchronization.

Reference

- Properties — 13 properties available on the component.
- Methods — 12 public methods.
- Events — 3 events you can handle.

TsgcWSCircuitBreaker

TsgcWSCircuitBreaker implements the Circuit Breaker resilience pattern for client-side calls to HTTP APIs.

Introduction

TsgcWSCircuitBreaker implements the Circuit Breaker resilience pattern for client-side calls to HTTP APIs. It prevents cascading failures when a remote endpoint becomes unavailable by tracking failures in a rolling window, opening the circuit once configurable thresholds are reached, and rejecting calls immediately during the cooldown period. It supports failure-count and failure-rate thresholds, slow-call detection, half-open trial calls, exception classification (record / ignore), per-endpoint overrides, fallback responses, and real-time metrics.

The circuit breaker integrates automatically with any descendant of **TsgcHTTPAPI_client** (OpenAI, Anthropic, Gemini, Grok, DeepSeek, Mistral, Ollama, Stripe, Binance, Bybit, Coinbase, Kraken and the other API wrappers) through the **CircuitBreaker** property. Once assigned, every Get / Post / Put / Patch / Delete call is wrapped automatically. One independent circuit is maintained per unique hostname.

Quick Start

1. Drop a **TsgcWSCircuitBreaker** component on the form.

2. Configure Thresholds (when to open) and Recovery (how to recover):

```
sgcWSCircuitBreaker1.Thresholds.FailureCount := 5;
sgcWSCircuitBreaker1.Recovery.CooldownSec := 30;
```

3. Assign the circuit breaker to an HTTP API client (any TsgcHTTPAPI_client descendant):

```
sgcAI_OpenAI1.CircuitBreaker := sgcWSCircuitBreaker1;
```

4. Call Get / Post / Delete / Patch / Put on the client. The breaker records successes and failures automatically and rejects new calls while the circuit is Open.

States

What it does: The circuit breaker is a three-state machine. Each host (key) has its own independent state that controls whether outbound calls are allowed, rejected, or sent as trials.

State	Behavior	Transition
csClosed	All calls pass through. Failures are counted in the rolling window.	Moves to csOpen when FailureCount, FailureRatePercent or SlowCallRatePercent is reached.
csOpen	Every call is rejected immediately. OnCallRejected fires for each attempt.	Moves to csHalfOpen after Recovery.CooldownSec seconds elapsed since opening.

csHal-fOpen	Up to Recovery.HalfOpenTrialCalls trial calls are allowed to probe recovery.	Moves to csClosed when the trials succeed; moves back to csOpen if any trial fails.

Top-level Properties

What it does: Master switch for the component plus the two key names used when no per-call key is supplied. The client-side integration uses the target hostname as the key automatically — DefaultKey and ServerKey only apply to manual Execute/RecordSuccess/RecordFailure calls and to the optional server-integration helpers.

When to use it: Disable the breaker at runtime during a controlled load test or during a maintenance window without unassigning it from the HTTP client. Set a meaningful DefaultKey (for example `'openai-api'`) when you drive Execute manually inside custom code.

Property	Type	Default	Description
Enabled	Boolean	True	Master switch. When False, every public method short-circuits to "allowed" and no counters are updated.
DefaultKey	string	'default'	Key used by the parameterless overloads of Execute, RecordSuccess and RecordFailure.
ServerKey	string	'server'	Key used by the optional server-integration hooks (IsConnectionAllowed, IsMessageAllowed, RecordMessageError).

```
sgcWSCircuitBreaker1->Enabled = true;
sgcWSCircuitBreaker1->DefaultKey = "openai-api";
sgcWSCircuitBreaker1->ServerKey = "server";
```

Thresholds

What it does: Defines the conditions that move the circuit from Closed to Open. The breaker opens when FailureCount raw failures are reached, or when FailureRatePercent of calls in the window fail (after MinCalls), or when SlowCallRatePercent of calls exceed SlowCallDurationMs.

When to use it: Your server calls OpenAI's API. When 5 failures occur within 60 seconds (or 50% of calls fail once at least 10 calls have been observed), open the circuit and stop hammering OpenAI — let the remote recover instead of piling on retries.

Property	Type	Default	Description
Enabled	Boolean	True	Enables threshold-based opening. When False the circuit never opens automatically.

FailureCount	Integer	5	Absolute number of failures in the rolling window that trips the breaker.
FailureRate-Percent	Integer	50	Failure percentage (0-100) that trips the breaker once MinCalls is reached.
SlowCallDura-tionMs	Integer	3000	Threshold (ms) above which a successful call is also counted as "slow".
SlowCallRate-Percent	Integer	0	Percentage (0-100) of slow calls that trips the breaker. 0 dis-ables slow-call detection.
MinCalls	Integer	10	Minimum calls observed before rate thresholds are evaluated (avoids tripping on small samples).

```
// Open after 5 failures, or 50% failure rate on at least 10 calls
sgcWSCircuitBreaker1->Thresholds->Enabled = true;
sgcWSCircuitBreaker1->Thresholds->FailureCount = 5;
sgcWSCircuitBreaker1->Thresholds->FailureRatePercent = 50;
sgcWSCircuitBreaker1->Thresholds->MinCalls = 10;
// Also treat any call slower than 3 seconds as a "slow" call
sgcWSCircuitBreaker1->Thresholds->SlowCallDurationMs = 3000;
sgcWSCircuitBreaker1->Thresholds->SlowCallRatePercent = 60;
```

TimeWindow

What it does: Defines the rolling-window used for the threshold evaluation. The window is divided into Bucket-Count sub-buckets so old counters age out smoothly as time advances. Thresholds are always evaluated against the sum of live buckets.

When to use it: Use a 30-second rolling window with 6 buckets so transient spikes don't trip the circuit but sus-tained failure does. Longer windows smooth noise; shorter windows react faster to outages. More buckets give smoother aging but consume slightly more memory per host.

Property	Type	Default	Description
RollingWin-dowSec	Inte-ger	60	Total width of the rolling window in seconds. Older samples are discarded.
BucketCount	Inte-ger	10	Number of sub-buckets the window is divided into. Each bucket holds success / failure / slow counters.

```
// 30-second window split into 6 buckets (5 seconds each)
sgcWSCircuitBreaker1->TimeWindow->RollingWindowSec = 30;
sgcWSCircuitBreaker1->TimeWindow->BucketCount = 6;
```

Recovery

What it does: Controls how the circuit recovers from the Open state. After CooldownSec elapsed, the breaker transitions to HalfOpen and allows HalfOpenTrialCalls probe calls through. If the trials succeed the breaker closes; if they fail it re-opens and the cooldown timer restarts.

When to use it: After opening, wait 30 seconds then let one trial call through to test recovery before re-closing. A longer cooldown protects a struggling upstream from another wave of traffic; a short cooldown reopens quickly for services that only flicker.

Property	Type	Default	Description
CooldownSec	Integer	30	Seconds to wait in the Open state before transitioning to HalfOpen.
HalfOpenTrialCalls	Integer	1	Number of trial calls permitted in HalfOpen before deciding to close or re-open.
AutoReset	Boolean	True	When True, counters are cleared automatically on state transitions.
MaxRetries	Integer	3	Informational ceiling consumed by retry helpers. No automatic retrying is done by the breaker itself.

```
// After opening: wait 30s, then allow 1 trial call, then decide
sgcWSCircuitBreaker1->Recovery->CooldownSec = 30;
sgcWSCircuitBreaker1->Recovery->HalfOpenTrialCalls = 1;
sgcWSCircuitBreaker1->Recovery->AutoReset = true;
sgcWSCircuitBreaker1->Recovery->MaxRetries = 3;
```

Fallback

What it does: Configures an alternative payload served while the circuit is Open. When Fallback.Enabled is True and Execute / ExecuteWithResult rejects a call, the OnFallback event fires with a response initialized from CachedResponse (or CustomMessage). Handlers may replace this string before it is returned to the caller.

When to use it: Return a cached last-known-good response when OpenAI is down so users see degraded-but-functional behavior (for example the previously generated summary) instead of an error. Use CustomMessage for a static "maintenance mode" JSON body.

Property	Type	Default	Description
Enabled	Boolean	False	Enables the fallback path. When False, rejected calls just return without firing OnFallback.

CachedResponse	string	''	Static payload returned while the circuit is Open. Takes precedence over CustomMessage.
CustomMessage	string	''	Fallback payload used when CachedResponse is empty.
UseLastSuccess	Boolean	False	When True, the last successful response observed for the key is served as the fallback.

```
// When api.openai.com is down, serve a safe JSON body instead of erroring
sgcWSCircuitBreaker1->Fallback->Enabled = true;
sgcWSCircuitBreaker1->Fallback->CachedResponse =
  "{\"error\":\"service unavailable\",\"retry_after\":30}";
sgcWSCircuitBreaker1->Fallback->CustomMessage =
  "{\"error\":\"ai service temporarily offline\"}";
sgcWSCircuitBreaker1->Fallback->UseLastSuccess = true;
```

Classification

What it does: Classifies raised exceptions as either "recorded as failure" or "ignored". Only recorded exceptions advance the failure counters. MatchMode selects how the strings in RecordAsFailure / IgnoreExceptions are compared to the exception text (`ClassName: Message`).

When to use it: Treat `EIdHTTPProtocolException` with 500/502/503/504 as failures, but ignore `EIdHTTPProtocolException` 400/401/404 (business errors should not trip the circuit). Leaving RecordAsFailure empty means every non-ignored exception is recorded.

Property	Type	Default	Description
RecordAsFailure	TStringList	empty	Patterns matched against exception text; matching exceptions count as failures. Empty list = record everything not ignored.
IgnoreExceptions	TStringList	empty	Patterns matched against exception text; matching exceptions are dropped (neither success nor failure).
MatchMode	TsgcCircuitBreakerExceptionMatch	cemContains	Comparison mode: cemExact, cemContains, cemStartsWith or cemWildcard (`*` / `?`).

```
// Record HTTP 5xx and timeouts as failures; ignore client-side 4xx
sgcWSCircuitBreaker1->Classification->MatchMode = cemContains;
sgcWSCircuitBreaker1->Classification->RecordAsFailure->Add("HTTP/1.1 500");
sgcWSCircuitBreaker1->Classification->RecordAsFailure->Add("HTTP/1.1 502");
sgcWSCircuitBreaker1->Classification->RecordAsFailure->Add("HTTP/1.1 503");
sgcWSCircuitBreaker1->Classification->RecordAsFailure->Add("HTTP/1.1 504");
sgcWSCircuitBreaker1->Classification->RecordAsFailure->Add("timeout");
sgcWSCircuitBreaker1->Classification->IgnoreExceptions->Add("HTTP/1.1 400");
sgcWSCircuitBreaker1->Classification->IgnoreExceptions->Add("HTTP/1.1 401");
```

```
sgcWSCircuitBreaker1->Classification->IgnoreExceptions->Add("HTTP/1.1 404");
```

PerEndpoint

What it does: Collection of pattern-based overrides. Each TsgcCircuitBreakerEndpointItem has its own Pattern (wildcards `*` and `?` supported), an OverrideThresholds block and an OverrideRecovery block. When the active key (hostname) matches an enabled item, that item's settings replace the top-level Thresholds / Recovery for the duration of the call.

When to use it: Separate circuits per API: `api.openai.com` has lenient thresholds (it occasionally slows under load), but `api.stripe.com` is payment-critical and should open faster to fail closed. Add one endpoint item per host with tuned thresholds.

Property	Type	Default	Description
Enabled	Boolean	False	Enables the endpoint-override collection. When False no items are evaluated.
Items	TCollection	empty	Collection of TsgcCircuitBreakerEndpointItem. First enabled match wins.

Each **TsgcCircuitBreakerEndpointItem** exposes:

Property	Type	Default	Description
Pattern	string	''	Wildcard pattern matched against the key (hostname). Supports `*` and `?`.
Enabled	Boolean	True	Disables this item without deleting it.
OverrideThresholds	TsgcCircuitBreakerThresholds	defaults	Thresholds applied when the pattern matches. Active only when its own Enabled is True.
OverrideRecovery	TsgcCircuitBreakerRecovery	defaults	Recovery settings applied when the pattern matches.

```
sgcWSCircuitBreaker1->PerEndpoint->Enabled = true;

// OpenAI - lenient: 10 failures, 60s cooldown
TsgcCircuitBreakerEndpointItem *oOpenAI = (TsgcCircuitBreakerEndpointItem*)
  sgcWSCircuitBreaker1->PerEndpoint->Items->Add();
oOpenAI->Pattern = "api.openai.com";
oOpenAI->Enabled = true;
oOpenAI->OverrideThresholds->Enabled = true;
oOpenAI->OverrideThresholds->FailureCount = 10;
oOpenAI->OverrideThresholds->FailureRatePercent = 70;
oOpenAI->OverrideRecovery->CooldownSec = 60;
```

```
// Stripe - payment critical: open fast, recover slow
TsgcCircuitBreakerEndpointItem *oStripe = (TsgcCircuitBreakerEndpointItem*)
  sgcWSCircuitBreaker1->PerEndpoint->Items->Add();
oStripe->Pattern = "api.stripe.com";
oStripe->Enabled = true;
oStripe->OverrideThresholds->Enabled = true;
oStripe->OverrideThresholds->FailureCount = 3;
oStripe->OverrideThresholds->FailureRatePercent = 30;
oStripe->OverrideRecovery->CooldownSec = 120;
```

Metrics

What it does: Read-only counters updated atomically as calls flow through the breaker. Aggregated across all keys / hostnames. CurrentOpenBreakers is refreshed after every state transition; AverageLatencyMs is a running average of the durations observed by Execute.

When to use it: Live dashboard showing per-host circuit states, failure rates and rejected counts. Poll Metrics from a TTimer to paint gauges for operators, or emit them to Prometheus / StatsD for alerting.

Property	Type	Description
TotalCalls	Int64	Total calls processed (successes + recorded failures).
TotalFailures	Int64	Total failures recorded across all keys.
TotalSuccesses	Int64	Total successful calls recorded across all keys.
TotalRejected	Int64	Total calls rejected because the circuit was Open.
CurrentOpen-Breakers	Inte-ger	Number of keys currently in the Open state.
AverageLatencyMs	Dou-ble	Running average latency (ms) measured around Execute / ExecuteWith-Result calls.

```
// Poll from a TTimer and update a dashboard
void __fastcall TForm1::MetricsTimer(TObject *Sender)
{
  LabelCalls->Caption = "Calls: " +
    IntToStr(sgcWSCircuitBreaker1->Metrics->TotalCalls) + " (ok " +
    IntToStr(sgcWSCircuitBreaker1->Metrics->TotalSuccesses) + " / fail " +
    IntToStr(sgcWSCircuitBreaker1->Metrics->TotalFailures) + " / rejected " +
    IntToStr(sgcWSCircuitBreaker1->Metrics->TotalRejected) + ")";
  LabelOpen->Caption = "Open breakers: " +
    IntToStr(sgcWSCircuitBreaker1->Metrics->CurrentOpenBreakers);
  LabelLatency->Caption = "Avg latency: " +
    FormatFloat("0.0", sgcWSCircuitBreaker1->Metrics->AverageLatencyMs) + " ms";
}
```

Events

OnStateChange: Fired when a circuit transitions between states. Provides the key, the old state and the new state (csClosed, csOpen or csHalfOpen). Useful for logging, alerting and updating live UI.

OnCallRejected: Fired when a call is rejected because the circuit is Open or the HalfOpen trial budget is exhausted. Provides the key and a reason string (for example `Circuit Open`).

OnFallback: Fired just before a fallback response is returned. The *aResponse* parameter (var string) is initialized from Fallback.CachedResponse / CustomMessage and may be replaced by the handler.

OnFailureRecorded: Fired every time a failure is recorded. Provides the key and the exception text (`ClassName: Message`). Use for structured logging or to correlate with upstream incident tracking.

OnSlowCall: Fired when a successful call exceeded Thresholds.SlowCallDurationMs. Provides the key and the measured duration in milliseconds.

Public Methods

Method	Description
Execute(Key, Action)	Runs the TProc under the given key. Checks IsCallAllowed first, records success / failure / slow automatically and returns True if the action executed.
Execute(Action)	Overload using DefaultKey.
ExecuteWithResult(Key, Action, out Result)	(D2009+) Runs a TFunc<TObject> and returns its result via the out parameter; same accounting as Execute.
RecordSuccess(Key)	Records a successful call. Advances HalfOpen -> Closed when trials succeed.
RecordSuccess	Overload using DefaultKey.
RecordFailure(Key, Exception)	Records a failure with optional exception text; triggers transition evaluation.
RecordFailure(Exception)	Overload using DefaultKey.
ForceOpen(Key)	Manually forces the circuit into the Open state.
ForceClose(Key)	Manually forces the circuit into the Closed state.
Reset(Key)	Clears counters and state for the given key.
ResetAll	Clears every tracked circuit and resets Metrics.
GetState(Key)	Returns the current TsgcCircuitState for the key.
IsCallAllowed(Key)	Returns True if a new call is currently allowed. Called automatically by the HTTP API client before every request.
GetFailureRate(Key)	Returns the current failure percentage (0.0 - 100.0) observed in the rolling window.

GetTotalCalls(Key)	Returns the number of calls observed in the rolling window for the key.
GetLastStateChange(Key)	Returns the TDateTime of the last state transition.
GetOpenBreakers	Returns TArray<string> of keys currently in the Open state.
SaveStateToFile(FileName)	Persists every circuit's state and counters to a file.
LoadStateFromFile(FileName)	Restores circuit states from a previously saved file.
IsConnectionAllowed(IP)	Server-side self-protection hook. Secondary to the primary client-side use. Returns True if the ServerKey circuit is not Open.
IsMessageAllowed(IP, Message)	Server-side self-protection hook. Returns True if the ServerKey circuit is not Open.
RegisterConnection(IP)	Server-side hook reserved for future per-IP metrics (currently no-op).
UnregisterConnection(IP)	Server-side hook reserved for future per-IP metrics (currently no-op).
RecordMessageError(IP, Exception)	Server-side self-protection hook. Records a failure on ServerKey.
RecordMessageSuccess(IP)	Server-side self-protection hook. Records a success on ServerKey.

Automatic Integration

When assigned to a TsgcHTTPAPI_client descendant via the CircuitBreaker property, the client automatically:

- **Before every Get / Post / Put / Patch / Delete:** Derives the key from the target URL hostname (for example `api.openai.com`) and calls IsCallAllowed(host). If the circuit is Open the call is rejected with the exception *"Circuit breaker open for <host> - call rejected"* and OnCallRejected fires.
- **On successful HTTP response:** Calls RecordSuccess(host) to update counters, enabling HalfOpen trials to close the circuit when the remote recovers.
- **On exception:** Calls RecordFailure(host, exception.Message), applies Classification rules to decide whether to count or ignore, evaluates transitions, then re-raises the exception to the caller.
- **One circuit per host:** Calls to different hosts (for example `api.openai.com`, `api.anthropic.com`, `api.stripe.com`) track fully independent circuits.

Complete Example

```
// Enable the circuit breaker
sgcWSCircuitBreaker1->Enabled = true;

// Thresholds: open after 5 failures, or 50% failure rate on 10+ calls
sgcWSCircuitBreaker1->Thresholds->Enabled = true;
sgcWSCircuitBreaker1->Thresholds->FailureCount = 5;
sgcWSCircuitBreaker1->Thresholds->FailureRatePercent = 50;
sgcWSCircuitBreaker1->Thresholds->SlowCallDurationMs = 3000;
sgcWSCircuitBreaker1->Thresholds->MinCalls = 10;
```

```
// Rolling window: 60 seconds / 10 buckets
sgcWSCircuitBreaker1->TimeWindow->RollingWindowSec = 60;
sgcWSCircuitBreaker1->TimeWindow->BucketCount = 10;

// Recovery: wait 30s in Open, then 1 trial call in HalfOpen
sgcWSCircuitBreaker1->Recovery->CooldownSec = 30;
sgcWSCircuitBreaker1->Recovery->HalfOpenTrialCalls = 1;

// Classification: record HTTP 5xx and timeouts, ignore 4xx
sgcWSCircuitBreaker1->Classification->MatchMode = cemContains;
sgcWSCircuitBreaker1->Classification->RecordAsFailure->Add("HTTP/1.1 5");
sgcWSCircuitBreaker1->Classification->RecordAsFailure->Add("timeout");
sgcWSCircuitBreaker1->Classification->IgnoreExceptions->Add("HTTP/1.1 404");

// Fallback: serve a cached payload while the circuit is Open
sgcWSCircuitBreaker1->Fallback->Enabled = true;
sgcWSCircuitBreaker1->Fallback->CustomMessage =
  "{\"error\":\"service unavailable\"}";

// Per-endpoint overrides
sgcWSCircuitBreaker1->PerEndpoint->Enabled = true;
TsgcCircuitBreakerEndpointItem *oStripe = (TsgcCircuitBreakerEndpointItem*)
  sgcWSCircuitBreaker1->PerEndpoint->Items->Add();
oStripe->Pattern = "api.stripe.com";
oStripe->OverrideThresholds->Enabled = true;
oStripe->OverrideThresholds->FailureCount = 3;
oStripe->OverrideRecovery->CooldownSec = 120;

// Persist circuit state across restarts
sgcWSCircuitBreaker1->LoadStateFromFile("circuit.dat");

// Attach to an HTTP API client (any TsgcHTTPAPI_client descendant)
sgcAI_OpenAI1->CircuitBreaker = sgcWSCircuitBreaker1;

// Use the client normally; the breaker intercepts Get / Post / etc.
try
{
  String Response = sgcAI_OpenAI1->Get("https://api.openai.com/v1/models");
}
catch (Exception &E)
{
  if (E.Message.Pos("Circuit breaker open") > 0)
    Log("Circuit open for api.openai.com - using fallback");
  else
    Log("Call failed: " + E.Message);
}

// Handle state changes
void __fastcall TForm1::CircuitStateChange(TObject *Sender,
  const String aKey, TsgcCircuitState aOldState, TsgcCircuitState aNewState)
{
  Log("Circuit " + aKey + ": " + IntToStr((int)aOldState) + " -> " +
    IntToStr((int)aNewState));
}

// Handle fallback responses
void __fastcall TForm1::CircuitFallback(TObject *Sender,
  const String aKey, String &aResponse)
{
  Log("Fallback used for " + aKey);
}
```

Thread Safety

The circuit breaker is fully thread-safe. All public methods use internal critical sections and thread-safe lists to protect concurrent access to per-key state, the rolling window and the metrics counters. A single component can be shared by multiple HTTP API clients and accessed from any thread (event handlers, timers, background workers) without external synchronization.

Reference

- Properties — 10 properties available on the component.

- Methods — 17 public methods.
- Events — 5 events you can handle.

TsgcWSAPIKeyManager

TsgcWSAPIKeyManager provides full-lifecycle management for API keys issued by sgcWebSockets servers.

Introduction

TsgcWSAPIKeyManager provides full-lifecycle management for API keys issued by sgcWebSockets servers. It generates cryptographically random keys with a configurable human-readable prefix, hashes them at rest with SHA-256, SHA-512 or Bcrypt, validates incoming requests by header or query string, rotates keys with a configurable grace period, revokes and renews keys, enforces scope-based authorization, maintains a persistent audit log and supports an optional IP allowlist. Keys are stored in memory or in an encrypted file that survives restarts.

The manager attaches to TsgcWebSocketHTTPServer and TsgcWSServer_HTTPAPI through the **APIKeyManager** property. Because the server's automatic hooks only see the peer IP, actual request authentication is performed by calling **IsRequestAuthorized** from the server's OnConnect event — this gives access to the full HTTP header block and the URL/query string so the configured **X-API-Key** header (or query parameter) can be parsed, validated and, if requested, matched against a required scope.

Quick Start

1. Drop a **TsgcWSAPIKeyManager** component on the form.

2. Configure Generation (prefix, length), Hashing (algorithm, salt) and Validation (header name, IP allowlist, Fail-Closed). See the per-section documentation below.

3. Assign the manager to a server component:

```
sgcWebSocketHTTPServer1.APIKeyManager := sgcWSAPIKeyManager1;
```

4. Wire the server's OnConnect to call **IsRequestAuthorized** using the connection's headers, URL and IP. Disconnect the client when it returns False.

5. Issue the first key for a customer and deliver the plaintext value to them exactly once — only the hashed form remains in storage afterwards.

Top-level Properties

What it does: Master switch for the whole component. When Enabled is False every public method short-circuits to "allowed" and the server auto-hooks become no-ops, so the key manager is effectively bypassed.

When to use it: Temporarily disable key enforcement during a controlled maintenance window or a staging deployment without unassigning the component from the server. Re-enable it when the change is done — no keys are lost, no scopes change, and the audit log keeps running the moment you flip it back on.

Property	Type	Default	Description
Enabled	Boolean	True	Master switch. When False, ValidateKey / IsRequestAuthorized always return True and server auto-hooks become no-ops.

```
sgcWSAPIKeyManager1->Enabled = true;
```

Generation

What it does: Controls how the plaintext key string is built by *IssueKey*. The key is a *KeyPrefix* followed by *KeyLength* random characters drawn from *Charset*. When IncludeChecksum is True a short checksum is appended so malformed keys can be rejected cheaply before a hash lookup.

When to use it: Branded keys for your SaaS — use `sgc_live_aB3…` in production and `sgc_test_…` on the staging cluster. The prefix makes it trivial for support staff to spot which environment a leaked key belongs to and for log scrubbers to redact it.

Property	Type	Default	Description
Enabled	Boolean	True	When False, IssueKey raises instead of generating a new key.
KeyPrefix	string	''	Human-readable prefix prepended to the random body (for example `sgc_live_`).
KeyLength	Integer	32	Number of random characters in the body of the key.
Charset	string	A-Za-z0-9	Alphabet used to generate the random body. Defaults to URL-safe alphanumerics when empty.
IncludeChecksum	Boolean	False	When True appends a short checksum suffix so obviously malformed keys can be rejected without a hash lookup.

```
// Branded production keys: sgc_live_aB3...
sgcWSAPIKeyManager1->Generation->Enabled = true;
sgcWSAPIKeyManager1->Generation->KeyPrefix = "sgc_live_";
sgcWSAPIKeyManager1->Generation->KeyLength = 32;
sgcWSAPIKeyManager1->Generation->IncludeChecksum = true;
```

Storage

What it does: Controls where hashed keys and the audit log live. *kstMemory* keeps everything in-process (cleared on shutdown). *kstFile* persists to *FileName*, optionally encrypted with *EncryptionKey*, and re-saves every *AutoSaveSeconds* seconds. *kstCustom* lets you implement your own SaveToFile / LoadFromFile hooks.

When to use it: Keep keys in memory during development for fast iteration. Switch to *kstFile* with *EncryptAtRest* := *True* for production so keys survive a server restart and are never exposed as plaintext on disk.

Property	Type	Default	Description
StorageType	TsgcAPIKeyStorageType	kstMemory	kstMemory (volatile), kstFile (persist to FileName) or kstCustom (user-provided SaveToFile / LoadFromFile).
FileName	string	''	File path used by SaveToFile / LoadFromFile when StorageType is kstFile.
EncryptAtRest	Boolean	False	When True encrypts the key store on disk using EncryptionKey.
EncryptionKey	string	''	Secret used to encrypt the persisted key file. Must be supplied when EncryptAtRest is True.
AutoSaveSeconds	Integer	0	When > 0 the manager re-saves the key store to disk every AutoSaveSeconds seconds.

```
// Production: encrypted file that survives restarts, auto-saved every 60s
sgcWSAPIKeyManager1->Storage->StorageType = kstFile;
sgcWSAPIKeyManager1->Storage->FileName = "apikeys.dat";
sgcWSAPIKeyManager1->Storage->EncryptAtRest = true;
sgcWSAPIKeyManager1->Storage->EncryptionKey = "my-very-secret-master-key";
sgcWSAPIKeyManager1->Storage->AutoSaveSeconds = 60;
sgcWSAPIKeyManager1->LoadFromFile("apikeys.dat");
```

Hashing

What it does: Plaintext keys are never persisted. IssueKey hashes the generated key using *Algorithm* with an optional *Salt* and *Iterations* count; ValidateKey rehashes the incoming value and performs a constant-time comparison against the stored digest.

When to use it: Never store plaintext. SHA-256 is the sensible default for most deployments. Use SHA-512 for higher-security profiles (FIPS 140-2 Level 3). Use Bcrypt when you need deliberate key-stretching — slower validation but dramatically harder offline brute-force of a leaked key store.

Property	Type	Default	Description
Algorithm	TsgcAPIKeyHashAlgorithm	khaSHA256	Hash function used for at-rest storage: khaSHA256, khaSHA512 or khaBcrypt.
Salt	string	''	Static salt mixed into every hash. Change invalidates all existing keys.
Iterations	Integer	1	Number of hashing rounds. Increase for key-stretching at the cost of validation latency.

```
// SHA-512 with a static salt and 10k iterations for key stretching
sgcWSAPIKeyManager1->Hashing->Algorithm = khaSHA512;
sgcWSAPIKeyManager1->Hashing->Salt = "my-salt";
sgcWSAPIKeyManager1->Hashing->Iterations = 10000;
```

Scopes

What it does: Catalog of allowed scope strings. When *Enabled* is True every scope attached to an issued key must exist in this list; unknown scopes are rejected. Individual catalog entries can be disabled without deleting them. ValidateKey / IsRequestAuthorized can require a specific scope per request.

When to use it: Fine-grained permissions. One customer's key carries `read:orders` only and will be rejected on a write endpoint. A partner integration receives `read:orders` + `write:shipments` but never `admin:*`. Each request presents a scope hint and the manager enforces it consistently.

Property	Type	Default	Description
Enabled	Boolean	True	Enables scope enforcement. When False, scope parameters are ignored.
Scopes	TCollection	empty	Collection of TsgcAPIKeyScopeItem entries defining the allowed scopes.

Each **TsgcAPIKeyScopeItem** exposes:

Property	Type	Default	Description
Name	string	''	Scope identifier (for example `read:orders`). Case-sensitive.
Description	string	''	Human-readable description shown in the object inspector and admin UIs.
Enabled	Boolean	True	Disables this scope without deleting it. Disabled scopes never satisfy HasScope.

```
sgcWSAPIKeyManager1->Scopes->Enabled = true;

TsgcAPIKeyScopeItem *scopeRead = (TsgcAPIKeyScopeItem*)
  sgcWSAPIKeyManager1->Scopes->Scopes->Add();
scopeRead->Name = "read:orders";
scopeRead->Description = "List and read order history";

TsgcAPIKeyScopeItem *scopeWrite = (TsgcAPIKeyScopeItem*)
  sgcWSAPIKeyManager1->Scopes->Scopes->Add();
scopeWrite->Name = "write:shipments";
scopeWrite->Description = "Create and update shipment records";
```

Rotation

What it does: Makes key rotation transparent. RotateKey issues a fresh key for the same owner and scopes and marks the old one *kksRotated*. During the *GracePeriodSec* window both keys validate so the customer can swap without downtime. When *AutoRotateDays* is > 0 keys older than that threshold are rotated by the background sweep; *NotifyBeforeExpirySec* fires OnKeyExpired early so you can email the customer.

When to use it: A customer reports their key was leaked but their integration is still live. Rotate the key: they receive a new one, the old one keeps working for 24 hours so they can roll it through their infrastructure, then the old one is invalidated automatically.

Property	Type	Default	Description
Enabled	Boolean	False	Enables rotation features (grace period and auto-rotate sweep).
GracePeriod-Sec	Integer	86400	Seconds the previous key stays valid after RotateKey. Default is 24 hours.
AutoRotate-Days	Integer	0	When > 0 the background sweep rotates keys older than this many days.
NotifyBefore-ExpirySec	Integer	604800	Fires OnKeyExpired this many seconds before a key actually expires. Default is 7 days.

```
// 24h grace period, auto-rotate keys older than 90 days, notify 7 days ahead
sgcWSAPIKeyManager1->Rotation->Enabled = true;
sgcWSAPIKeyManager1->Rotation->GracePeriodSec = 86400;
sgcWSAPIKeyManager1->Rotation->AutoRotateDays = 90;
sgcWSAPIKeyManager1->Rotation->NotifyBeforeExpirySec = 7 * 86400;
```

Expiration

What it does: Controls how issued keys age out. *DefaultTTLSec* is applied to IssueKey when the caller passes 0 as ExpiresInSec. *EnforceExpiry* toggles whether expired keys are rejected. *CheckIntervalSec* controls how often the background sweep scans for expired entries, fires OnKeyExpired and updates Stats.

When to use it: Partner integration keys must expire after 90 days — partners have to re-sign the agreement and fetch a new key. Setting DefaultTTLSec to 90×86400 means every IssueKey call respects that policy unless the caller explicitly overrides it.

Property	Type	Default	Description
DefaultT-TLSec	Integer	0	Default time-to-live for newly issued keys in seconds. 0 means never expires.
EnforceExpiry	Boolean	True	When True ValidateKey rejects keys whose ExpiresAt is in the past.

CheckInterValSec	Integer	60	How often the background sweep scans for expired keys and fires OnKeyExpired.

```
// Partner keys expire after 90 days, enforced strictly, swept every minute
sgcWSAPIKeyManager1->Expiration->DefaultTTLSec = 90 * 86400;
sgcWSAPIKeyManager1->Expiration->EnforceExpiry = true;
sgcWSAPIKeyManager1->Expiration->CheckIntervalSec = 60;
```

RateLimit

What it does: Stores per-key rate-limit defaults (max requests per window) alongside each key. These values are informational hints that a TsgcWSRateLimiter instance can read through the PerAPIKey scope to apply tier-appropriate throttling — the API key manager itself does not throttle traffic.

When to use it: Tiered SaaS. Free-tier customers are provisioned at 100 req/hour, Pro-tier at 10,000 req/hour, Enterprise at 1,000,000 req/hour. Configure DefaultMaxRequests / DefaultWindowSec on the API key manager and pair it with TsgcWSRateLimiter.PerAPIKey so the rate limiter reads the per-key budget automatically.

Property	Type	Default	Description
Enabled	Boolean	False	Enables per-key rate-limit metadata alongside each issued key.
DefaultMaxRequests	Integer	100	Default maximum requests per window assigned to new keys.
DefaultWindowSec	Integer	60	Default window size in seconds associated with DefaultMaxRequests.

```
// Default tier: 100 requests per hour per API key
sgcWSAPIKeyManager1->RateLimit->Enabled = true;
sgcWSAPIKeyManager1->RateLimit->DefaultMaxRequests = 100;
sgcWSAPIKeyManager1->RateLimit->DefaultWindowSec = 3600;
// Pair with TsgcWSRateLimiter.PerAPIKey for actual throttling
```

Audit

What it does: Records every key-lifecycle action (issued, validated, revoked, rotated, expired, scope granted/revoked, validation failed) to the in-memory ring buffer and, when *LogFile* is set, appends the same entries to a file. *IncludeIP* records the requester's IP; *IncludePayload* captures additional details. *RetentionDays* prunes older entries; *MaxMemoryEntries* caps the ring buffer.

When to use it: Compliance requirement: every key operation must be logged and retained for 12 months. Enable audit with LogFile pointing at your SIEM drop zone and RetentionDays set to 365. OnAuditEvent fires for every entry so you can also ship them to a message bus.

Property	Type	Default	Description
Enabled	Boolean	True	Enables the audit log. When False no entries are recorded and OnAuditEvent never fires.
LogFile	string	''	Optional file path the audit log is appended to.
IncludeIP	Boolean	True	When True every entry records the requester's IP.
IncludePayload	Boolean	False	When True the Details field of each audit entry carries additional context.
RetentionDays	Integer	90	Entries older than this many days are pruned by the background sweep.
MaxMemoryEntries	Integer	10000	Maximum number of entries kept in the in-memory ring buffer.

```
// Compliance: 12-month retention, log to file, include IP + payload
sgcWSAPIKeyManager1->Audit->Enabled = true;
sgcWSAPIKeyManager1->Audit->LogFile = "apikeys-audit.log";
sgcWSAPIKeyManager1->Audit->IncludeIP = true;
sgcWSAPIKeyManager1->Audit->IncludePayload = true;
sgcWSAPIKeyManager1->Audit->RetentionDays = 365;
sgcWSAPIKeyManager1->Audit->MaxMemoryEntries = 50000;
```

Validation

What it does: Tells the manager how clients present their keys and under what conditions requests are accepted. *HeaderName* (default `X-API-Key`) and *QueryParamName* (default `api_key`) are the two places ExtractKeyFromHeaders / ExtractKeyFromQuery look. *RequireHTTPS* rejects plaintext traffic. *IPAllowlist* restricts keys to specific source IPs. *FailClosed* (default True) rejects any request the manager cannot positively authorize.

When to use it: Corporate-only API: require HTTPS for every request and restrict keys to the office egress CIDR range so even a leaked key cannot be abused from outside the corporate network.

Property	Type	Default	Description
HeaderName	string	X-API-Key	HTTP header name used to read the incoming key.
QueryParamName	string	api_key	Query-string parameter used as fallback when the header is absent.
RequireHTTPS	Boolean	False	When True rejects every non-HTTPS request regardless of key validity.
IPAllowlist	TStringList	empty	When non-empty, only requests whose IP appears in the list are authorized.

FailClosed	Boolean	True	When True any request the manager cannot positively authorize is rejected.

```
// Corporate API: HTTPS-only, allow only office egress IPs, fail closed
sgcWSAPIKeyManager1->Validation->HeaderName = "X-API-Key";
sgcWSAPIKeyManager1->Validation->QueryParamName = "api_key";
sgcWSAPIKeyManager1->Validation->RequireHTTPS = true;
sgcWSAPIKeyManager1->Validation->FailClosed = true;
sgcWSAPIKeyManager1->Validation->IPAllowlist->Add("203.0.113.10");
sgcWSAPIKeyManager1->Validation->IPAllowlist->Add("203.0.113.11");
```

Stats

What it does: Read-only counter object that exposes live statistics about the key store. All counters are updated atomically inside the component's internal critical section.

When to use it: Live admin dashboard. Poll Stats from a TTimer and paint the current ActiveKeys / RevokedKeys / TotalValidations / TotalValidationFailures so operators can see authentication pressure in real time.

Property	Type	Description
TotalKeys	Int64	Total number of keys ever issued by this manager instance.
ActiveKeys	Int64	Number of keys currently in status kksActive.
RevokedKeys	Int64	Number of keys currently in status kksRevoked.
ExpiredKeys	Int64	Number of keys currently in status kksExpired.
TotalValidations	Int64	Total number of ValidateKey / IsRequestAuthorized calls.
TotalValidationFailures	Int64	Total number of validation calls that rejected the key.
LastIssuedAt	TDateTime	Timestamp of the most recent IssueKey call.

```
// Poll from a TTimer and update a dashboard
void __fastcall TForm1::StatsTimer(TObject *Sender)
{
  LabelActive->Caption = "Active: " + IntToStr(sgcWSAPIKeyManager1->Stats->ActiveKeys);
  LabelRevoked->Caption = "Revoked: " + IntToStr(sgcWSAPIKeyManager1->Stats->RevokedKeys);
  LabelValidations->Caption = "Validations: " + IntToStr(sgcWSAPIKeyManager1->Stats->TotalValidations);
  LabelFailures->Caption = "Failures: " + IntToStr(sgcWSAPIKeyManager1->Stats->TotalValidationFailures);
}
```

Events

OnKeyIssued: Fired when a new key has been issued. Provides the owner, the raw plaintext key (last chance to observe it) and the scope array. Deliver the plaintext to the customer from this handler.

OnKeyValidated: Fired every time ValidateKey completes. Provides the key, a boolean *aValid* flag and a machine-readable reason string.

OnKeyRevoked: Fired when a key has been revoked. Provides the key and the revocation reason.

OnKeyRotated: Fired when a key has been rotated. Provides the old key and the new key so you can notify the customer.

OnKeyExpired: Fired by the background expiry sweep when a key's ExpiresAt has elapsed (or NotifyBeforeExpirySec seconds earlier).

OnValidation: Final decision hook fired during ValidateKey. Set *Allow* := False and provide a *Reason* to reject an otherwise-valid key (custom business rules).

OnAuditEvent: Fired for every audit log entry. Provides the TsgcAPIKeyAuditEntry record (Timestamp, KeyPrefix, Action, IP, Details) — ideal for forwarding to an external SIEM.

Public Methods

Method	Description
IssueKey(Owner, Scopes, ExpiresInSec)	Generates, hashes and stores a new key; returns the plaintext (only time it can be seen). Pass ExpiresInSec = 0 to inherit Expiration.DefaultTTLSec.
ValidateKey(Key, RequiredScope, IP)	Validates a raw key. Optionally enforces a required scope and records the requester's IP. Returns True on success.
RevokeKey(Key, Reason)	Marks a key as revoked; subsequent ValidateKey calls return False.
RotateKey(OldKey, out NewKey)	Issues a new key for the same owner and scopes; the old key is marked kksRotated and honored during GracePeriodSec.
RenewKey(Key, NewExpiresInSec)	Extends the key's expiration by the given number of seconds from now.
GetKeyInfo(Key)	Returns a TsgcAPIKeyInfo record with owner, scopes, status, timestamps and usage count.
GetKeysByOwner(Owner)	Returns hashed-key identifiers associated with the given owner.
ListScopes(Key)	Returns the scopes attached to a key.
IsExpired(Key)	Returns True if the key is past its expiry.
IsRevoked(Key)	Returns True if the key is revoked.
KeyExists(Key)	Returns True if the key is known (active, revoked or expired).
GrantScope(Key, Scope)	Adds a scope to an existing key.

RevokeScope(Key, Scope)	Removes a scope from an existing key.
HasScope(Key, Scope)	Returns True if the key carries the given scope.
GetAuditLog(Key, MaxEntries)	Returns audit entries; when Key is empty, returns entries for all keys.
ClearAuditLog(Key)	Clears the audit log for a specific key or (when empty) for all keys.
SaveToFile(FileName)	Persists keys and audit log to the given file (falls back to Storage.FileName when empty).
LoadFromFile(FileName)	Restores keys and audit log from file.
ExportKeys(Stream)	Stream-based export of keys and audit log.
ImportKeys(Stream)	Stream-based import of keys and audit log.
PurgeExpired	Removes all keys whose ExpiresAt is in the past.
Count	Returns the number of keys currently stored.
IsRequestAuthorized(Headers, QueryString, IP, RequiredScope)	One-shot authorization check: extracts the key from headers or query string, validates it and optionally enforces a scope.
ExtractKeyFromHeaders(Headers)	Parses the Validation.HeaderName value out of a raw HTTP header block.
ExtractKeyFromQuery(QueryString)	Parses the Validation.QueryParamName value out of a raw query string.
IsConnectionAllowed(IP)	Server hook: returns False when FailClosed is True and IPAllowlist is non-empty and IP is not listed. Called automatically.
IsMessageAllowed(IP, Message)	Server hook called automatically for every inbound message. IP-level no-op by default.
RegisterConnection(IP)	Tracks a new connection. Called automatically by the server.
UnregisterConnection(IP)	Releases tracking for a connection. Called automatically on disconnect.

Automatic Integration

When assigned to a server's APIKeyManager property, the component automatically:

- **On new connection:** The server calls IsConnectionAllowed(IP) which enforces the optional IPAllowlist when FailClosed is True (otherwise it is a no-op). RegisterConnection(IP) is invoked to track the live connection.
- **On message received:** The server calls IsMessageAllowed(IP, Message), also an IP-level no-op unless the allowlist rejects the IP. These hooks *do not* inspect HTTP headers, so API-key authentication is not performed here.
- **Request authorization (user must wire):** Call *IsRequestAuthorized(Connection.HeadersRequest.Text, Connection.URL, Connection.IP, RequiredScope)* from the server's OnConnect event. This is the point

where the raw HTTP headers are available, so this is where the X-API-Key header is extracted, validated and optionally scope-checked. Disconnect the connection when the function returns False.

- **Client side:** Set the X-API-Key header (or the configured HeaderName) from the client's OnHandshake / OnBeforeConnect event so the server can read it during the HTTP upgrade.
- **On disconnect:** UnregisterConnection(IP) is called to release internal tracking.

Complete Example

```
// Enable and configure the API key manager
sgcWSAPIKeyManager1->Enabled = true;

// Branded keys
sgcWSAPIKeyManager1->Generation->KeyPrefix = "sgc_live_";
sgcWSAPIKeyManager1->Generation->KeyLength = 32;

// Persist to an encrypted file that survives restarts
sgcWSAPIKeyManager1->Storage->StorageType = kstFile;
sgcWSAPIKeyManager1->Storage->FileName = "apikeys.dat";
sgcWSAPIKeyManager1->Storage->EncryptAtRest = true;
sgcWSAPIKeyManager1->Storage->EncryptionKey = "master-secret";
sgcWSAPIKeyManager1->Storage->AutoSaveSeconds = 60;

// SHA-256 with a static salt
sgcWSAPIKeyManager1->Hashing->Algorithm = khaSHA256;
sgcWSAPIKeyManager1->Hashing->Salt = "my-salt";

// Scope catalog
sgcWSAPIKeyManager1->Scopes->Enabled = true;
TsgcAPIKeyScopeItem *scopeRead = (TsgcAPIKeyScopeItem*)
  sgcWSAPIKeyManager1->Scopes->Scopes->Add();
scopeRead->Name = "read:orders";
scopeRead->Description = "Read orders";
TsgcAPIKeyScopeItem *scopeWrite = (TsgcAPIKeyScopeItem*)
  sgcWSAPIKeyManager1->Scopes->Scopes->Add();
scopeWrite->Name = "write:orders";
scopeWrite->Description = "Create orders";

// 90-day default TTL and 24-hour rotation grace period
sgcWSAPIKeyManager1->Expiration->DefaultTTLSec = 90 * 86400;
sgcWSAPIKeyManager1->Rotation->Enabled = true;
sgcWSAPIKeyManager1->Rotation->GracePeriodSec = 86400;

// Full audit log, 12-month retention
sgcWSAPIKeyManager1->Audit->Enabled = true;
sgcWSAPIKeyManager1->Audit->LogFile = "apikeys-audit.log";
sgcWSAPIKeyManager1->Audit->IncludeIP = true;
sgcWSAPIKeyManager1->Audit->RetentionDays = 365;

// Validation: header-based, fail closed
sgcWSAPIKeyManager1->Validation->HeaderName = "X-API-Key";
sgcWSAPIKeyManager1->Validation->QueryParamName = "api_key";
sgcWSAPIKeyManager1->Validation->FailClosed = true;

// Load previously saved keys and attach to server
sgcWSAPIKeyManager1->LoadFromFile("apikeys.dat");
sgcWebSocketHTTPServer1->APIKeyManager = sgcWSAPIKeyManager1;
sgcWebSocketHTTPServer1->Active = true;

// Issue a new key for a customer (plaintext returned only here)
DynamicArray<String> scopes;
scopes.Length = 2;
scopes[0] = "read:orders";
scopes[1] = "write:orders";
String LKey = sgcWSAPIKeyManager1->IssueKey("customer-123", scopes, 30 * 86400);
String LNewKey;
ShowMessage("Deliver to customer: " + LKey);

// Authorize every incoming connection in OnConnect
void __fastcall TForm1::ServerConnect(TsgcWSConnection *Connection)
{
  if (!sgcWSAPIKeyManager1->IsRequestAuthorized(
        Connection->HeadersRequest->Text,
        Connection->URL, Connection->IP, "read:orders"))
    Connection->Disconnect();
}

// Rotate a leaked key (old one valid 24h during grace)
sgcWSAPIKeyManager1->RotateKey(LKey, LNewKey);
```

```
// Revoke a key on subscription cancellation
sgcWSAPIKeyManager1->RevokeKey(LNewKey, "customer cancelled subscription");
```

Thread Safety

The API key manager is fully thread-safe. All public methods use internal critical sections and thread-safe lists to protect concurrent access to the key store, scope index, owner index and audit log. A single component can be shared by multiple server instances and accessed from any thread (server event handlers, timer threads, background workers) without external synchronization.

Reference

- Properties — 10 properties available on the component.
- Methods — 26 public methods.
- Events — 7 events you can handle.

TsgcWSCluster

TsgcWSCluster enables multi-node clustering so a publish or Presence event on one server node reaches subscribers connected to another node, through a pluggable backplane.

Introduction

TsgcWSCluster enables multi-node clustering for sgcWebSockets servers. A single server process can only deliver a published message to the clients connected to that same process. When you scale out behind a load balancer, a client subscribed to a channel on node A never receives a message published on node B. TsgcWSCluster solves this by forwarding publishes and Presence events between nodes over a backplane, so the existing Publish / channel / Presence APIs keep working unchanged across the whole cluster.

Clustering is a **server feature** and requires the **Enterprise** edition. The backplane is pluggable through the **EngineType** property: **clusterMesh** (the default) needs zero external infrastructure, while **clusterRedis** reuses an existing Redis server as the transport. Once a protocol server is attached, every message published on any node is replicated to the other nodes automatically without any manual event wiring.

Quick Start

1. Drop a **TsgcWSCluster** component on the form, together with a server and a **TsgcWSPServer_sgc** protocol component.

2. Choose the engine, the listener port and the peers (mesh engine):

3. Attach the protocol server so its publishes are clustered, then start the cluster and the server:

```
sgcWSCluster1->EngineType = clusterMesh;
sgcWSCluster1->ClusterPort = 5410;
sgcWSCluster1->Peers->Add("192.168.1.101:5410");
sgcWSCluster1->Peers->Add("192.168.1.102:5410");
sgcWSCluster1->Attach(sgcWSPServer_sgc1);
sgcWSCluster1->Start();
sgcWebSocketServer1->Active = true;
```

Engine Types

What it does: EngineType selects the backplane used to move messages between nodes. **clusterMesh** needs no external service: every node opens a listener on its own *ClusterPort* and dials the addresses listed in *Peers*, forming a direct full mesh. **clusterRedis** publishes and subscribes on a shared Redis channel, so nodes never connect to each other directly — Redis is the single hub.

When to use it: Pick **clusterMesh** for a small, fixed set of nodes where you do not want to run any extra infrastructure. Pick **clusterRedis** when you already operate a Redis server, when the node set changes frequently, or when you want a central point that every node trusts.

Value	Infrastructure	Description

clusterMesh	None	Each node listens on ClusterPort and connects to the addresses in Peers. Default engine. Available in Delphi, C++Builder and .NET.
clusterRedis	Redis server	Nodes exchange messages through a shared Redis Pub/Sub channel. Delphi and C++Builder only in this release.

Note (.NET): the .NET implementation supports the Mesh engine only. Setting EngineType to clusterRedis raises a NotSupported error.

Attaching Protocols

What it does: Attach connects a server-side pub/sub protocol to the cluster so its publishes and Presence events are replicated to the other nodes. Attach the sgc protocol server (**TsgcWSPServer_sgc**) to cluster channel messages, and optionally the Presence server (**TsgcWSPServer_Presence**) to make Presence cluster-wide. In .NET assign the *Protocol* and/or *Server* properties instead of calling Attach.

```
sgcWSCluster1->Attach(sgcWSPServer_sgc1);       // cluster channel publishes
sgcWSCluster1->Attach(sgcWSPServer_Presence1);  // cluster-wide Presence
```

Properties

Property	Type	Default	Description
Enabled	Boolean	False	Master switch. When False the cluster does not start and no messages are replicated.
EngineType	TsgcWSClusterEngineType	clusterMesh	Backplane engine: clusterMesh (zero infrastructure) or clusterRedis (Redis Pub/Sub, Delphi only).
NodeId	string	''	Unique node identifier. A GUID is generated automatically on Start when left empty.
ClusterPort	Integer	5410 (Delphi) / 5414 (.NET)	Mesh listener port for this node. Each node connects to the peers on their own ClusterPort.
Peers	TStringList	empty	Peer nodes as "host:port" lines, one per line. Used by the mesh engine.
RedisHost	string	'127.0.0.1'	Redis server host. Used when EngineType is clusterRedis. Delphi only.
RedisPort	Integer	6379	Redis server port. Used when EngineType is clusterRedis. Delphi only.
RedisPassword	string	''	Optional Redis AUTH password. Used when EngineType is clusterRedis. Delphi only.

Redis-Channel	string	'sgccluster'	Redis Pub/Sub channel shared by every node in the cluster. Delphi only.
Connect-edPeer-Count	Integer	—	Read-only. Number of peer links currently connected.
Active	Boolean	—	Read-only. True while the cluster backplane is started.
Ready	Boolean	—	Read-only. True when the configured peers are connected, or when no peers are configured.

Events

OnConnect: Fired when the cluster backplane has started.

OnDisconnect: Fired when the cluster backplane has stopped.

OnPeerConnected: Fired when a peer link comes up. Provides the peer address as "host:port".

OnPeerDisconnected: Fired when a peer link goes down. Provides the peer address as "host:port".

OnClusterMessage: Fired when a message arrives from another node. Provides the TsgcWSClusterMessage that was received.

Note (.NET): OnPeerConnected and OnPeerDisconnected are exposed as `Action<string>`, and an additional **OnClusterError** event reports backplane errors.

Public Methods

Method	Description
Start	Starts the cluster backplane (opens the mesh listener and dials the peers, or subscribes to the Redis channel).
Stop	Stops the cluster backplane and drops every peer link.
Attach(aRoutable)	Attaches a server-side pub/sub protocol (the sgc protocol server, and optionally the Presence server) so its publishes and Presence events are clustered.
Detach(aRoutable)	Detaches a protocol previously attached with Attach, stopping replication of its messages.

Complete Example

```
// Node A (192.168.1.101) - mesh engine
sgcWSCluster1->EngineType = clusterMesh;
sgcWSCluster1->NodeId = "node-a";
sgcWSCluster1->ClusterPort = 5410;
sgcWSCluster1->Peers->Add("192.168.1.102:5410");

// Cluster channel publishes and cluster-wide Presence
sgcWSCluster1->Attach(sgcWSPServer_sgc1);
sgcWSCluster1->Attach(sgcWSPServer_Presence1);

// React to peer links coming up and down
sgcWSCluster1->OnPeerConnected = ClusterPeerConnected;
sgcWSCluster1->OnPeerDisconnected = ClusterPeerDisconnected;

// Start the cluster, then the server
sgcWSCluster1->Start();
sgcWebSocketServer1->Active = true;

// A publish on this node now reaches subscribers on node-b too
sgcWSPServer_sgc1->WriteData("A message for every node", "news");

void __fastcall TForm1::ClusterPeerConnected(TObject *Sender, const String aPeer)
{
  Log("Peer connected: " + aPeer);
}

void __fastcall TForm1::ClusterPeerDisconnected(TObject *Sender, const String aPeer)
{
  Log("Peer disconnected: " + aPeer);
}
```

Requirements

TsgcWSCluster is a server component available in the **Enterprise** edition. It works with the existing Publish, channel and Presence APIs unchanged: the only change needed to make an existing single-node server cluster-aware is to drop the component, attach the protocol server and start it. The mesh engine adds no external dependencies; the Redis engine requires a reachable Redis server.

Thread Safety

The cluster backplane runs on its own connections and synchronizes access to the peer list and routing tables with internal critical sections. The public methods can be called from server event handlers, timer threads or background workers without external synchronization.

Reference

- Properties — 12 properties available on the component.
- Methods — 4 public methods.
- Events — 5 events you can handle.

Guides

- **Clustering**
  - Cluster Mesh Engine
  - Cluster Redis Engine
  - Cluster Presence

Cluster: Mesh Engine

The mesh engine is the default TsgcWSCluster backplane and needs no external infrastructure. Each node opens a listener on its own ClusterPort and connects to the addresses listed in Peers, forming a direct full mesh. A publish or Presence event on any node is then replicated to every other node, so subscribers connected to a different node still receive it.

How It Works

1. Each node is given a unique NodeId (auto-generated as a GUID on Start when empty) and a ClusterPort to listen on.
2. Each node lists the other nodes it should dial in the Peers property, as "host:port" lines, where the port is the remote node's ClusterPort.
3. On Start, the node opens its listener and connects to its peers. Links are bidirectional, so one established connection between two nodes is enough.
4. Once a protocol server is attached, every message it publishes is forwarded to the connected peers and delivered to their local subscribers. The originating NodeId is used to discard echoes and prevent loops.

Properties

Property	Description
EngineType	Set to clusterMesh (the default).
NodeId	Unique node identifier. Auto-generated as a GUID on Start when empty.
ClusterPort	The port this node listens on for peers. Default 5410 (Delphi/C++Builder), 5414 (.NET).
Peers	The other nodes to connect to, one "host:port" per line.

Configuring a Node

Set the engine, the port and the peers, attach the protocol server, then start the cluster and the server:

```
sgcWSCluster1->EngineType = clusterMesh;
sgcWSCluster1->ClusterPort = 5410;
sgcWSCluster1->Peers->Add("192.168.1.101:5410");
sgcWSCluster1->Peers->Add("192.168.1.102:5410");
sgcWSCluster1->Attach(sgcWSPServer_sgc1);
sgcWSCluster1->Start();
sgcWebSocketServer1->Active = true;
```

Three-Node Mesh

In a three-node cluster each node lists the other two as peers. Because links are bidirectional, listing every other node on every node is fine; duplicate links are collapsed into a single connection.

```
// Node A (192.168.1.101)
sgcWSCluster1->NodeId = "node-a";
sgcWSCluster1->ClusterPort = 5410;
sgcWSCluster1->Peers->Add("192.168.1.102:5410");
sgcWSCluster1->Peers->Add("192.168.1.103:5410");
```

Monitoring Peers

Use ConnectedPeerCount and the OnPeerConnected / OnPeerDisconnected events to track the health of the mesh. Ready returns True once the configured peers are connected (or immediately when no peers are configured).

```
sgcWSCluster1->OnPeerConnected = ClusterPeerConnected;
sgcWSCluster1->OnPeerDisconnected = ClusterPeerDisconnected;

void __fastcall TForm1::ClusterPeerConnected(TObject *Sender, const String aPeer)
{
  Log("Peer connected: " + aPeer);
}

void __fastcall TForm1::ClusterPeerDisconnected(TObject *Sender, const String aPeer)
{
  Log("Peer disconnected: " + aPeer);
}
```

Note: the mesh engine is available in Delphi, C++Builder and .NET. The ClusterPort must be reachable between nodes; it is independent of the WebSocket port the server exposes to clients.

Cluster: Redis Engine

The Redis engine uses an existing Redis server as the cluster backplane. Instead of connecting nodes directly to each other, every node publishes to and subscribes from a shared Redis Pub/Sub channel, and Redis fans the messages out. This is a good fit when you already operate Redis, when the set of nodes changes frequently, or when you want a single central hub that every node trusts.
Note: the Redis engine is available in Delphi and C++Builder. The .NET implementation supports the Mesh engine only; setting EngineType to Redis in .NET raises a NotSupported error.

How It Works

1. Every node is configured with EngineType = clusterRedis and points at the same Redis server (RedisHost / RedisPort) and the same channel (RedisChannel).
2. On Start, each node subscribes to RedisChannel and opens a publishing connection to Redis.
3. When a protocol server attached to the cluster publishes a message, the node serializes it and publishes it to RedisChannel.
4. Redis delivers the message to every subscribed node, which delivers it to its local subscribers. The originating NodeId is used to discard the echo on the sender.

Properties

Property	Description
EngineType	Set to clusterRedis.
RedisHost	Redis server host. Default 127.0.0.1.
RedisPort	Redis server port. Default 6379.
RedisPassword	Optional Redis AUTH password. Leave empty when the server has no password.
RedisChannel	Shared Pub/Sub channel. Default sgccluster. Nodes sharing the same channel form one cluster.

Configuring a Node

Point the node at the Redis server, attach the protocol server, then start the cluster and the server. ClusterPort and Peers are not used by the Redis engine.

```
sgcWSCluster1->EngineType = clusterRedis;
sgcWSCluster1->RedisHost = "127.0.0.1";
sgcWSCluster1->RedisPort = 6379;
sgcWSCluster1->RedisChannel = "sgccluster";
sgcWSCluster1->Attach(sgcWSPServer_sgc1);
sgcWSCluster1->Start();
sgcWebSocketServer1->Active = true;
```

Authentication

When the Redis server requires authentication, set RedisPassword. The password is sent with the Redis AUTH command when the backplane connects.

```
sgcWSCluster1->EngineType = clusterRedis;
sgcWSCluster1->RedisHost = "10.0.0.50";
sgcWSCluster1->RedisPort = 6379;
sgcWSCluster1->RedisPassword = "my-redis-secret";
```

Multiple Clusters on One Redis Server

Nodes that share the same RedisChannel belong to the same cluster. Use distinct channel names to run several independent clusters against a single Redis server, for example to separate a production cluster from a staging cluster.

```
// Production nodes
sgcWSCluster1->RedisChannel = "cluster-production";

// Staging nodes (same Redis server, isolated cluster)
sgcWSCluster1->RedisChannel = "cluster-staging";
```

Cluster: Cluster-Wide Presence

Presence lets clients see who else is connected and subscribed. On a single server, Presence covers only the clients connected to that server. When you attach the Presence server to TsgcWSCluster, Presence becomes cluster-wide: a join or leave on one node is replicated to the other nodes, so every client sees the combined roster of the whole cluster.

How It Works

1. Drop the sgc protocol server (TsgcWSPServer_sgc) and the Presence server (TsgcWSPServer_Presence) on the form, alongside the server and the TsgcWSCluster component.
2. Attach both protocols to the cluster: the sgc protocol server for channel publishes, and the Presence server for cluster-wide Presence.
3. When a client joins or leaves on any node, the Presence event is replicated to the other nodes so their clients see the change.
4. The existing Presence API is unchanged; attaching the Presence server is the only step needed to make it span the cluster.

Attaching the Presence Server

Attach the sgc protocol server and the Presence server, then start the cluster and the server:

```
sgcWSCluster1->EngineType = clusterMesh;
sgcWSCluster1->ClusterPort = 5410;
sgcWSCluster1->Peers->Add("192.168.1.102:5410");

// Channel publishes and cluster-wide Presence
sgcWSCluster1->Attach(sgcWSPServer_sgc1);
sgcWSCluster1->Attach(sgcWSPServer_Presence1);

sgcWSCluster1->Start();
sgcWebSocketServer1->Active = true;
```

Using Presence Across Nodes

Once attached, the regular Presence and channel APIs work cluster-wide without any further change. A client subscribed to a channel on node A receives the join, leave and publish events produced by clients on node B.

```
// A publish on this node reaches subscribers on every node in the cluster
sgcWSPServer_sgc1->WriteData("Hello cluster", "lobby");
```

Note: attach the Presence server in addition to the sgc protocol server. Attaching only the sgc protocol server clusters channel publishes but leaves Presence local to each node.

TsgcWebSocketLoadBalancerServer

TsgcWebSocketLoadBalancerServer component.

Introduction

The component **TsgcWebSocketLoadBalancerServer** allows you to load-balance **WebSocket** and **HTTP** protocols. For the WebSocket protocol, it distributes messages across a group of servers and distributes client connections using a random sequence or fewest-connections algorithm.

The Load Balancer Server inherits all methods and properties from TsgcWebSocketHTTPServer.

Load Balancer Configuration

The Load Balancer server is a descendant of TsgcWebSocketHTTPServer, so read the documentation about the TsgcWebSocketHTTPServer to know how to configure it.

Additionally, the Load Balancer has the property LoadBalancer, which has the following properties:

- **LoadBalancing**: configure here how to distribute the connections
  - **lbRandom:** (default) every time a new client requests a connection, it will return a random server.
  - **lbConnections:** every time a new client requests a connection, it will return the server with the fewest connected clients.
- Protocols: configure which protocols are enabled
  - **WebSocket:** if true, the websocket connections will be handled by the Load Balancer Server.
  - **HTTP:** if true, the http connections will be handled by the Load Balancer Server.

Backup Server Configuration

The Backup Servers (the servers behind the load balancer) can be a TsgcWebSocketServer, TsgcWebSocketHTTPServer or a Datasnap Server.
Those servers have a property called **LoadBalancer** where you can configure the connection between the LoadBalancer Server and the Backup Servers.

- **Enabled:** set to true if you want to use as a backup server.
- **Host:** the host where the LoadBalancer is located.
- **Port:** the listening port of the LoadBalancer.
- **Guid:** unique id that identifies this server.
- **Bindings:** the public addresses where the connections will be forwarded. Example: if the Backup WebSocket server is listening on port 8000 and the ip address is 1.1.1.1, use the following: ws://1.1.1.1:8000;
- **AutoRegisterBindings:** if enabled, the LoadBalancer Server will use the Bindings property of the backup server to configure the public bindings.
- **AutoRestart:** in seconds, if greater than zero, the load balancer client of the backup server will enable an internal watchdog that every x seconds, will check if the connection is alive, if it's closed, it will try to reconnect.

Reference

- Properties — 18 properties available on the component.
- Methods — 9 public methods.
- Events — 22 events you can handle.

TsgcWebSocketProxyServer

TsgcWebSocketProxyServer implements a WebSocket Server Component that listens for client WebSocket connections and forwards data connections to a normal TCP/IP server.

Introduction

TsgcWebSocketProxyServer implements a WebSocket Server Component that listens for client WebSocket connections and forwards data connections to a normal TCP/IP server. This is especially useful for browser connections because it allows a browser to virtually connect to any server.

Reference

- Properties — 19 properties available on the component.
- Methods — 7 public methods.
- Events — 22 events you can handle.

TsgcIWWebSocketClient

TsgcIWWebSocketClient implements an IntraWeb WebSocket Component that can connect to a WebSocket Server.

Introduction

TsgcIWWebSocketClient implements an IntraWeb WebSocket Component that can connect to a WebSocket Server. Follow the steps below to configure this component:

1. Drop a TsgcIWWebSocketClient component in the form

2. Set Host and Port (default is 80) to connect to an available WebSocket Server. You can set URL property and Host, Port, Parameters... will be updated from URL. Example: wss://127.0.0.1:8080/ws/ will result:

```
oClient = new TsgcIWWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClient->TLS = true;
oClient->Options->Parameters = "/ws/";
```

3. You can select if you want TLS (secure connection) or not, by default it is not activated.

4. Set Transports allowed.

WebSockets: it will use standard WebSocket implementation

Emulation: if browser doesn't support WebSockets, then it will use a loop AJAX callback connection

6. Create an Async Procedure and set property Active := True

Methods

Open: Opens a WebSocket Connection.

Close: Closes a WebSocket Connection.

WriteData: sends a message to WebSocket Server.

Properties

Connected: is a read-only variable and returns True if the connection is Active, otherwise returns False.

JSOpen: here you can include JavaScript Code on the client side when a connection is opened.

JSClose: here you can include JavaScript Code on the client side when a connection is closed.

JSMessage: here you can include JavaScript Code on the client side when clients receive a message from the server. You can get Message String, using Javascript variable "text".

JSError: here you can include JavaScript Code on the client side when an error is raised. You can get Message Error, using Javascript variable "text".

Reference

- Properties — 11 properties available on the component.
- Methods — 4 public methods.
- Events — 5 events you can handle.

TsgcWSConnection

TsgcWSConnection is a wrapper for client WebSocket connections. You can access this object in Server or Client Events.

Methods

WriteData: sends a message to the client.

WriteSSEEvent: writes one server-sent event to a connection whose Transport is trpSSE: an "event:" line with the event name (omitted when empty), an "id:" line with the id (omitted when empty), one "data:" line per line of the data, and the blank line that ends the event. CR and LF are removed from the event name and the id. The whole event is written under the critical section of the connection, so it never interleaves with another write, and it is queued as one item when a text queue is configured. It does nothing on any other transport.

WriteSSEComment: writes a comment line (": text") and the blank line to a trpSSE connection. Browsers ignore a comment, so it is the way to keep an idle stream alive through a proxy. Comments are never queued.

Close: sends a close message to other peer. A "CloseCode" can be specified optionally. By default, the value sent is NORMAL close code. If you send a negative close code, the reason for closing will not be sent.

Disconnect: close client connection from the server side. A "CloseCode" can be specified optionally.

Ping: sends a ping to the client.

AddTCPEndOfFrame: if connection is plain TCP, allows you to set which byte/s define the end of message. Message is buffered till is received completely.

Subscribed: returns if the connection is subscribed to a custom channel.

Subscribe: subscribe this connection to a channel. Later you can Broadcast a message from server component to all connections subscribed to this channel.

UnSubscribe: unsubscribe this connection from a channel.

Properties

Protocol: returns sub-protocol used on this connection.

IP: returns Peer IP Address.

Port: returns Peer Port.

LocalIP: returns Host IP Address.

LocalPort: returns Host Port.

URL: returns URL requested by the client.

Guid: returns connection ID.

HeadersRequest: returns a list of Headers received on Request.

HeadersResponse: returns a list of Headers sent as Response.

RecBytes: number of bytes received.

SendBytes: number of bytes sent.

PendingCount: number of messages still queued for this connection, added across the three priority levels of QueueOptions. Returns 0 when nothing is queued, and reading it allocates nothing. Use it to run a credit window on top of the OnQueueDrained event, or to detect a consumer that is falling behind. This property is read-only, and it is available on the managed .NET connection with the same meaning.

Transport: returns the transport type of connection:

trpRFC6455: a normal WebSocket connection.

trpHixie76: a WebSocket connection using draft WebSocket spec.

trpSSE: a Server-Sent Events connection.

trpTCP: plain TCP connection.

TCPEndOfFrameScanBuffer: allows defining which method use to find end of message (if using trpTCP as transport).

eofScanNone: every time a new packet arrives, the OnBinary event is called.

eofScanLatestBytes: if latest bytes are equal to bytes added with AddTCPEndOfFrame method, OnBinary message is called, otherwise this packet is buffered

eofScanAllBytes: searches in the entire packet for bytes equal to bytes added with the AddTCPEndOfFrame method. If found, the OnBinary event is called, otherwise this packet is buffered

Data: user session data object, here you can pass an object and access this every time you need, for example: you can pass a connection to a database, user session properties...

Protocols

With WebSockets, you can implement sub-protocols, allowing you to create customized communications. **For example,** you can implement a sub-protocol over the WebSocket protocol to communicate with a customized application using JSON messages, and you can implement another sub-protocol using XML messages.

When a connection is opened on the Server side, it will validate if the sub-protocol sent by the client is supported by the server; if not, it will close the connection. A server can implement several sub-protocols, but only one can be used on a single connection.

Sub-protocols are very useful for creating customized applications and ensuring that all clients support the same communication interface.

Although the protocol name is arbitrary, it's recommended to use unique names like "dataset.esegece.com"

With sgcWebSockets package, you can build your own protocols and you can use built-in sub-protocols provided:

1. Protocol MQTT: MQTT is a Client Server publish/subscribe messaging transport protocol. It is lightweight, open, simple, and designed so as to be easy to implement.

2. Protocol WebRTC: open source project aiming to enable the web with Real-Time Communication (RTC) capabilities.

3. Protocol Files: implemented using binary messages, provides support for sending files: packet size, authorization, QoS, message acknowledgement and more.

4. Protocol SGC: implemented using JSON-RPC 2.0 messages, provides the following patterns: RPC, PubSub, Transactional Messages, Messages Acknowledgment and more.

5. Protocol Dataset: inherits from Default Protocol, can send dataset changes (new record, save record or delete a record) from the server to clients.

6. Protocol Presence: allows you to know who is subscribed to a channel, example: chat rooms, collaborators on a document, people viewing the same web page, competitors in a game...

7. Protocol WAMP 1.0: open WebSocket subprotocol that provides two asynchronous messaging patterns: RPC and PubSub.

8. Protocol WAMP 2.0: open WebSocket subprotocol that provides two asynchronous messaging patterns: RPC and PubSub.

9. Protocol STOMP: STOMP is the Simple (or Streaming) Text Orientated Messaging Protocol. STOMP provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers.

9.1 STOMP for RabbitMQ: client for RabbitMQ Broker.

9.2 STOMP for ActiveMQ: client for ActiveMQ Broker.

10. Protocol AMQP: Advanced Message Queuing Protocol (AMQP 0.9.1) is created as an open standard protocol that allows messaging interoperability between systems, regardless of message broker vendor or platform used.

11. Protocol AMQP1: Advanced Message Queuing Protocol (AMQP 1.0.0) is created as an open standard protocol that allows messaging interoperability between systems, regardless of message broker vendor or platform used.

12. Protocol E2EE: Messages sent over a WebSocket connection are encrypted end-to-end at the application level, so only the communicating clients can read them even though the WebSocket server relays the data.

If you need to use **more than one protocol using a single connection** (example: you may need to use **default protocol** to handle Remote Procedure Calls and **Dataset protocol** to handle database connections) you can as-

sign a "Broker" to each protocol component and all messages will be exchanged using this intermediary protocol (you can check "Tickets Demo" to get a simple example of this).

Protocols can be registered at **runtime**, just call Method **RegisterProtocol** and pass protocol component as a parameter.

Javascript Reference

Here you can get more information about the common JavaScript library used in sgcWebSockets.

Protocols Javascript

Default Javascript sgcWebSockets uses **sgcWebSocket.js** file.

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure your access to sgcWebSocket.js file as:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
</script>
```

sgcWebSocket has 3 parameters, only first is required:

```
sgcWebSocket(url, protocol, transport)
```

- **URL:** WebSocket server location, you can use "ws:" for normal WebSocket connections and "wss:" for secured WebSocket connections.

  ```
  sgcWebSocket('ws://127.0.0.1')

  sgcWebSocket('wss://127.0.0.1')
  ```

- **Protocol:** if the server accepts one or more protocol, you can define which protocol you want to use.

  ```
  sgcWebSocket('ws://127.0.0.1', 'esegece.com')
  ```

- **Transport:** by default, first tries to connect using WebSocket connection and if not implemented by Browser, then tries Server Sent Events as Transport.

  Use WebSocket if implemented, if not, then use Server Sent Events:

  ```
  sgcWebSocket('ws://127.0.0.1')
  ```

  Only use WebSocket as transport:

  ```
  sgcWebSocket('ws://127.0.0.1', '', ['websocket'])
  ```

  Only use Server Sent as transport:

  ```
  sgcWebSocket('ws://127.0.0.1', '', ['sse'])
  ```

Open Connection With Authentication

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket({"host":"ws://{%host%}:{%port%}","user":"admin","password":"1234"});
</script>
```

Send Message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
  socket.send('Hello sgcWebSockets!');
</script>
```

Show Alert with Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
  socket.on('message', function(event)
  {
    alert(event.message);
  }
</script>
```

Binary Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
  socket.on('stream', function(event)
  {
    document.getElementById('image').src = URL.createObjectURL(event.stream);
    event.stream = "";
  }
</script>
```

Binary (Header + Image) Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
  socket.on('stream', function(event)
  {
    sgcWSStreamRead(evt.stream, function(header, stream) {
      document.getElementById('text').innerHTML = header;
      document.getElementById('image').src = URL.createObjectURL(event.stream);
      event.stream = "";
    }
  }
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
```

```
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error: ' + event.message);
  };
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  socket.close();
</script>
```

Get Connection Status

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  socket.state();
</script>
```

Protocol MQTT

MQTT is a Client-Server publish/subscribe messaging transport protocol. It is lightweight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and the Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium.

The protocol runs over TCP/IP, or over other network protocols that provide ordered, lossless, bi-directional connections. Its features include:

· Use of the publish/subscribe message pattern which provides one-to-many message distribution and decoupling of applications.

· A messaging transport that is agnostic to the content of the payload.

· Three qualities of service for message delivery:
  · "At most once", where messages are delivered according to the best efforts of the operating environment. Message loss can occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.
  · "At least once", where messages are assured to arrive but duplicates can occur.
  · "Exactly once", where messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.

· A small transport overhead and protocol exchanges minimized to reduce network traffic.

· A mechanism to notify interested parties when an abnormal disconnection occurs.

Features

- Supports **3.1.1** and **5.0** MQTT versions.
- **Publish/subscribe** message pattern to provide one-to-many message distribution and decoupling of applications.
- **Acknowledgment** of messages sent.
- Implements **QoS** (Quality of Service) for message delivery (all levels: At most once, At least once and Exactly once)
- **Last Will Testament**.
- **Secure** connections.
- **HeartBeat** and **Watchdog**.
- **Authentication** to server.

Components

TsgcWSPClient_MQTT: MQTT Client Component.

Most common uses

- **Connection**
  - Client MQTT Connect
  - Connect Mosquitto MQTT Servers
  - Client MQTT Sessions
  - Client MQTT Version

- **Publish & Subscribe**
  - MQTT Publish Subscribe
  - MQTT Topics
  - MQTT Subscribe
  - MQTT Publish Message

- MQTT Receive Messages
- MQTT Publish and Wait Response

- **Other**
  - MQTT Clear Retained Messages

TsgcWSPClient_MQTT

The MQTT component provides a lightweight, fully-featured MQTT client implementation with support for versions 3.

Introduction

The MQTT component provides a lightweight, fully-featured MQTT client implementation with support for versions 3.1.1 and 5.0. The component supports plaintext and secure connections over both standard TCP and WebSockets.

Connection to an MQTT server is simple, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property. Set host and port in TsgcWebSocketClient and set Active := True to connect.

MQTT v5.0 is not backward compatible (like v3.1.1). Obviously too many new things are introduced so existing implementations have to be revisited.

According to the specification, MQTT v5.0 adds a significant number of new features to MQTT while keeping much of the core in place.

- The Clean Session flag functionality is divided into 2 properties to allow for finer control over session state data: the CleanStart parameter and the new SessionExpInterval.
- Server disconnect: Allow DISCONNECT to be sent by the Server to indicate the reason the connection is closed.
- All response packets (CONNACK, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBACK, UNSUBACK, DISCONNECT) now contain a reason code and reason string describing why operations succeeded or failed.
- Enhanced authentication: Provide a mechanism to enable challenge/response style authentication including mutual authentication. This allows SASL style authentication to be used if supported by both Client and Server, and includes the ability for a Client to re-authenticate within a connection.
- The Request / Response pattern is formalized by the addition of the ResponseTopic.
- Shared Subscriptions: Add shared subscription support allowing for load balanced consumers of a subscription.
- Topic Aliases can be sent by both client and server to refer to topic filters by shorter numerical identifiers in order to save bandwidth.
- Servers can communicate what features they support in ConnectionProperties.
- Server reference: Allow the Server to specify an alternate Server to use on CONNACK or DISCONNECT. This can be used as a redirect or to do provisioning.
- More: message expiration, Receive Maximums and Maximum Packet Sizes, and a Will Delay interval are all supported.

Reference

- Properties — 10 properties available on the component.
- Methods — 9 public methods.
- Events — 13 events you can handle.

TsgcWSPClient_MQTT | Client MQTT Connect

In order to connect to a MQTT Server, you must create first a TsgcWebSocketClient and a TsgcWSPClient_MQTT. Then you must attach MQTT Component to WebSocket Client.

Basic Usage

Connect to Mosquitto MQTT server using websocket protocol. Subscribe to topic: "topic1" after connect.

```
oClient = new TsgcWebSocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8080;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;

void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const int ReasonCode,
  const string ReasonName, const TsgcWSMQTTCONNACKProperties *ConnectProperties);
{
  oMQTT->Subscribe("topic1");
}
```

Client Identifier

MQTT requires a **Client Identifier** to identify client connection. Component sets a **random value** automatically but you can set your own Client Identifier if required, to do this, just handle **OnBeforeConnect** event and set your value on aClientIdentifier parameter.

```
void OnMQTTBeforeConnect(TsgcWSConnection *Connection, ref bool aCleanSession,
  ref string aClientIdentifier)
{
  aClientIdentifier = "your client id";
}
```

Authentication

Some servers require a user and password to a**uthorize MQTT connections**. Use **Authentication** property to set the value for username and password before connect to server.

```
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Authentication->Enabled = true;
oMQTT->Authentication->UserName = "your user";
oMQTT->Authentication->Password = "your password";
```

TsgcWSPClient_MQTT | Connect MQTT Mosquitto

Use the following sample configurations to connect to a Mosquitto MQTT Server.

MOSQUITTO MQTT WebSockets

```
oClient = new TsgcWebSocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8080;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;
```

MOSQUITTO MQTT WebSockets TLS

```
oClient = new TsgcWebSocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8081;
oClient->TLS = true;
oClient->TLSOptions->Version = tls1_2;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;
```

MOSQUITTO MQTT Plain TCP

```
oClient = new TsgcWebSocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 1883;
oClient->Specifications->RFC6455 = false;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;
```

MOSQUITTO MQTT Plain TCP TLS

```
oClient = new TsgcWebSocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8083;
oClient->Specifications->RFC6455 = false;
oClient->TLS = true;
oClient->TLSOptions->Version = tls1_2;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;
```

TsgcWSPClient_MQTT | Client MQTT Sessions

Clean Start

OnMQTTBeforeConnect event, there is a parameter called **aCleanSession**. If the value of this parameter is **True**, means that the client **wants to start a new session**, so if server has any session stored, it must discard it. So, when OnMQTTConnect event is fired, aSession parameter will be false. If the value of this parameter is **False** and there is a session associated to this client identifier, the server must resume communications with the client on state with the existing session.

So, if client has an **unexpected disconnection**, and you want to **recover the session** where was disconnected, in OnMQTTBeforeConnect set **aCleanSession = True** and **aClientIdentifier = Client ID of Session**.

Session

Once successful connection, check **OnMQTTConnect** event, the value of Session parameter.

Session = true, means session has been resumed.
Session = false, means it's a new session.

```
void OnMQTTBeforeConnect(TsgcWSConnection *Connection, ref bool aCleanSession,
  ref string aClientIdentifier)
{
  aCleanSession = false;
  aClientIdentifier = "previous client id";
}

void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const int ReasonCode,
  const string ReasonName, const TsgcWSMQTTCONNACKProperties *ConnectProperties);
{
  if (Session == true)
  {
    WriteLn("Session resumed");
  }
  else
  {
    WriteLn("New Session");
  }
}
```

TsgcWSPClient_MQTT | Client MQTT Version

Currently, MQTT Client supports the following specifications:

- **MQTT 3.1.1:** http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
- **MQTT 5.0:** https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

You can select which is the version which will use the MQTT Client component using MQTTVersion property.

MQTT 3.1.1: TsgcWSPClient_MQTT.Version = mqtt311
MQTT 5.0: sgcWSPClient_MQTT.Version = mqtt5

TsgcWSPClient_MQTT | MQTT Publish Subscribe

The publish/subscribe pattern (also known as pub/sub) provides an alternative to traditional client-server architecture. In the client-sever model, a client communicates directly with an endpoint.The pub/sub model **decouples the client that sends a message (the publisher) from the client or clients that receive the messages (the subscribers)**. The publishers and subscribers never contact each other directly. In fact, they are not even aware that the other exists. **The connection between them is handled by a third component (the broker)**. The job of the broker is to filter all incoming messages and distribute them correctly to subscribers.

With **TsgcWSPClient_MQTT** you can **Publish messages** and **Subscribe to Topics**.

Subscribe Topic

Subscribe to Topic "topic1" after a successful connection.

```
oClient = new TsgcWebSocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8080;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;

void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const int ReasonCode,
  const string ReasonName, const TsgcWSMQTTCONNACKProperties *ConnectProperties);
{
  oMQTT->Subscribe("topic1");
}
```

Publish Message

Publish a message to all subscribers of "topic1"

```
oClient = new TsgcWebSocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8080;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;

void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const int ReasonCode,
  const string ReasonName, const TsgcWSMQTTCONNACKProperties *ConnectProperties);
{
  oMQTT->Publish("topic1", "Hello Subscribers topic1");
}
```

TsgcWSPClient_MQTT | MQTT Topics

Topics

In MQTT, the word topic refers to an UTF-8 string that the broker uses to filter messages for each connected client. The topic consists of one or more topic levels. Each topic level is separated by a forward slash (topic level separator)

myHome / groundfloor / livingroom / temperature

In comparison to a message queue, MQTT topics are very lightweight. The client does not need to create the desired topic before they publish or subscribe to it. The broker accepts each valid topic without any prior initialization. Note that each topic must contain at least 1 character and that the topic string permits empty spaces. Topics are case-sensitive.

WildCards

When a client subscribes to a topic, it can subscribe to the exact topic of a published message or it can use wildcards to subscribe to multiple topics simultaneously. A wildcard can only be used to subscribe to topics, not to publish a message. There are two different kinds of wildcards: _single-level and _multi-level.

Single Level: +

As the name suggests, a single-level wildcard replaces one topic level. The plus symbol represents a single-level wildcard in a topic.

myHome / groundfloor / + / temperature

Any topic matches a topic with single-level wildcard if it contains an arbitrary string instead of the wildcard. For example a subscription to _myhome/groundfloor/+/temperature can produce the following results:

YES => myHome / groundfloor / livingroom / temperature
YES => myHome / groundfloor / kitchen / temperature
NO => myHome / groundfloor / livingroom / brightness
NO => myHome / firstfloor / livingroom / temperature
NO => myHome / groundfloor / kitchen / fridge / temperature

Multi Level:

The multi-level wildcard covers many topic levels. The hash symbol represents the multi-level wild card in the topic. For the broker to determine which topics match, the multi-level wildcard must be placed as the last character in the topic and preceded by a forward slash.

myHome / groundfloor / #

YES => myHome / groundfloor / livingroom / temperature
YES => myHome / groundfloor / kitchen / temperature
YES => myHome / groundfloor / kitchen / brightness
NO => myHome / firstfloor / kitchen / temperature

When a client subscribes to a topic with a multi-level wildcard, it receives all messages of a topic that begins with the pattern before the wildcard character, no matter how long or deep the topic is. If you specify only the multi-level wildcard as a topic (_#), you receive all messages that are sent to the MQTT broker.

TsgcWSPClient_MQTT | MQTT Subscribe

You can Subscribe to a Topic using method Subscribe from TsgcWSPClient_MQTT. This method has the following parameters:

Topic: is the name of the topic to be subscribed.
QoS: one of the 3 QoS levels (not all brokers support all 3 levels). If not specified, uses mtqsAtMostOnce. Read more about QoS Levels.
SubscribeProperties: if MQTT 5.0, are additional properties about subscriptions.

Subscribe QoS = At Least Once

```
MQTT->Subscribe("topic1", mtqsAtLeastOnce);
```

Subscribe MQTT 5.0

```
oProperties = new TsgcWSMQTTSubscribe_Properties();
oProperties->SubscriptionIdentifier = 1234;
oProperties->UserProperties->Add("name=value");

MQTT->Subscribe("topic1", mtqsAtMostOnce, oProperties);
```

TsgcWSPClient_MQTT | MQTT Publish Message

You can publish messages to all subscribers of a Topic using **Publish** method, which has the following parameters:

Topic: is the name of the topic where the message will be published.
Text: is the text of the message.
QoS: one of the 3 QoS levels (not all brokers support all 3 levels). If not specified, uses mtqsAtMostOnce. Read more about QoS Levels.
Retain: if true, this message will be retained. And every time a new client subscribes to this topic, this message will be sent to this client.
PublishProperties: if MQTT 5.0, these are the properties of the message.

Publish a simple message

```
MQTT->Publish("topic1", "Hello Subscribers topic1");
```

Publish QoS = At Least Once

```
MQTT->Publish("topic1", "Hello Subscribers topic1", mtqsAtLeastOnce);
```

Publish Retained message

```
MQTT->Publish("topic1", "Hello Subscribers topic1", mtqsAtMostOnce, true);
```

TsgcWSPClient_MQTT | MQTT Receive Messages

Messages sent by the server are received in the **OnMQTTPublish** event. This event has the following parameters:

Topic: is the name of the topic associated to this message.
Text: is the text of the message.
PublishProperties: if MQTT 5.0, these are the properties of the published message.

Read published Messages

```
void OnMQTTPublish(TsgcWSConnection *Connection, string aTopic, string aText,
  TsgcWSMQTTPublishProperties *PublishProperties)
{
  WriteLn("Topic: " + aTopic + ". Message: " + aText);
}
```

TsgcWSPClient_MQTT | MQTT Receive Messages (Extended)

The **OnMQTTPublishEx** event provides the published message payload in multiple formats through a **TsgcWSMQTTPublishData** object. This event has the following parameters:

Topic: is the name of the topic associated to this message.
Data: contains the payload of the published message. It has the following properties:

- **Value:** the payload as a string.
- **Bytes:** the raw payload as TBytes.
- **Stream:** the raw payload as a TMemoryStream.

PublishProperties: if MQTT 5.0, these are the properties of the published message.

Read published Messages (Extended)

```
void OnMQTTPublishEx(TsgcWSConnection *Connection, string aTopic,
  TsgcWSMQTTPublishData *aData, TsgcWSMQTTPublishProperties *PublishProperties)
{
  // read as string
  WriteLn("Topic: " + aTopic + ". Message: " + aData->Value);
  // read as bytes
  WriteLn("Bytes Length: " + IntToStr(aData->Bytes.Length));
  // read as stream
  WriteLn("Stream Size: " + IntToStr(aData->Stream->Size));
}
```

TsgcWSPClient_MQTT | Publish and Wait Response

MQTT client allows the use of some type of QoS levels, any of those levels works in a different level to be sure that messages have been processed as expected.

There are the following QoS levels:

- **mtqsAtMostOnce:** (by default) the message is delivered according to the best efforts of the underlying TCP/IP network. A response is not expected and no retry semantics are defined in the protocol. The message arrives at the server either once or not at all.
- **mtqsAtLeastOnce:** the receipt of a message by the server is acknowledged by an ACKNOWLEDGMENT message. If there is an identified failure of either the communications link or the sending device or the acknowledgement message is not received after a specified period of time, the sender resends the message. The message arrives at the server at least once. A message with QoS level 1 has an ID param in the message.
- **mtqsExactlyOnce:** where messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied. If there is an identified failure of either the communications link or the sending device, or the acknowledgement message is not received after a specified period of time, the sender resends the message.

You can handle the events OnPubAck or OnPubComp to know if the message has been processed by server or you can use the method **PublishAndWait** to know if the message has been processed by the server.
The use of **PublishAndWait** is the same as the normal Publish method, but now you have a new parameter called Timeout, where the method will return false if after a certain period of time, there is no response from server. By default this value is 10 seconds.

```
if mqtt->PublishAndWait("topic", "text")
{
  ShowMessage("Message processed")
}
else
{
  ShowMessage("Message error");
}
```

TsgcWSPClient_MQTT | MQTT Clear Retained Messages

By default, every MQTT topic can have a retained message. The standard MQTT mechanism to clean up retained messages is sending a retained message with an empty payload to a topic. This will remove the retained message.

```
MQTT->Publish("topic1", "", mtqsAtMostOnce, true);
```

Protocol AMQP 0.9.1

The **Advanced Message Queuing Protocol** (AMQP) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns. It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).

Features

AMQP can be used in any situation if there is a need for high-quality and secure message delivery between client and broker.

AMQP provides the following features:

- Monitoring and sharing updates.
- Ensuring quick response of the server to requests and transmission of time-consuming tasks for further processing.
- Distribute messages to multiple recipients.
- Connection offline clients for further data retrieval.
- Increase the reliability and smooth operation of applications.
- Reliability of message delivery.
- High speed message delivery.
- Message Acceptance.

Components

TsgcWSPClient_AMQP: it's the client component that implements **AMQP 0.9.1** protocol.

Most common uses

- **Connection**
  - Client AMQP Connect
  - Client AMQP Disconnect
- **Commands**
  - AMQP Channels
  - AMQP Exchanges
  - AMQP Queues
  - AMQP Publish Messages
  - AMQP Consume Messages (Asynchronous)
  - AMQP Get Messages (Synchronous)
  - AMQP QoS
  - AMQP Transactions

TsgcWSPClient_AMQP

AMQP 0.

Introduction

The **TsgcWSClient_AMQP** client implements the full **AMQP 0.9.1** protocol following the OASIS specification. The client supports Plain TCP and WebSocket connections, TLS (secure) connections are supported too.

Connection

AMQP 0.9.1 protocol defines the concept of channels, which allows you to share a single socket connection with several virtual channels, the client implements an internal thread which reads the bytes received and dispatch every message to the correct channel (which already runs in its own thread), so, if you are running an AMQP connection with 5 channels, the client will run 6 threads (5 threads which handle the data of every channel and 1 thread which handles the data of the connection).

Before connecting to an AMQP server, configure the following properties of the AMQP protocol

- **AMQPOptions.Locale:** it's the message locale to use, it's a negotiated value, so can change when compared with the supported locales supported by the server. The default value is "en_US".
- **AMQPOptions.MaxChannels:** it's the maximum number of channels which can be opened, it's a negotiated value, so can change when compared with the server configuration. The default value is 65535.
- **AMQPOptions.MaxFrameSize:** it's the maximum size in bytes of the AMQP frame, it's a negotiated value, so can change when compared with the server configuration. The default value is 2147483647.
- **AMQPOptions.VirtualHost:** it's the name of the virtual host. The default value is "/".

The AMQP HeartBeat can be configured too before connecting to the server, you can enable or disable the use of heartbeats.

- **HeartBeat.Enabled:** set to true if the client supports HeartBeats.
- **HeartBeat.Interval:** the desired interval in seconds.

Once the AMQP client has been configured, attach to a TsgcWebSocketClient and now you can configure the server connection properties to connect to the AMQP Server.
Set the property value **Specifications.RFC6455** to false if using Plain TCP connection instead of WebSocket connection.

```
TsgcWSPClient_AMQP *oAMQP = new TsgcWSPClient_AMQP();
oAMQP->AMQPOptions->Locale = "en_US";
oAMQP->AMQPOptions->MaxChannels = 100;
oAMQP->AMQPOptions->MaxFrameSize = 16384;
oAMQP->AMQPOptions->VirtualHost = "/";
oAMQP->HeartBeat->Enabled = true;
oAMQP->HeartBeat->Interval = 60;

TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
oAMQP->Client = oClient;
oClient->Specifications->RFC6455 = false;
oClient->Host = "www.esegece.com";
oClient->Port = 5672;
oClient->Active = true;
```

Channels

Once the AMQP client has connected, it can open the first channel.

```
oAMQP->OpenChannel("channel_name");
```

Exchanges

When a Channel is opened, the client can declare new exchanges, verify that they exist... use the method **Declare-Exchange** to declare a new exchange.

```
oAMQP->DeclareExchange("channel_name", "exchange_name");
```

Queues

When a Channel is opened, the client can declare new queues, verify that they exist... use the method Declare-Queue to declare a new Queue. The queues are not provided by default by the server (unlike the exchanges), so it's always required to declare a new queue (unless a queue has been already created by another client).

```
oAMQP->DeclareQueue("channel_name", "queue_name");
```

Binding Queues

Once the Exchanges and Queues are configured, you may need to bind queues to exchanges, this way the exchanges can know which messages will be dispatched to the queues.

AMQP Servers automatically bind the queues to "direct" exchange using the queue name as routing key. This allows you to send a message to a specific queue without the need to declare a binding (just calling PublishMessage method and passing the Exchange argument as empty value and the name of the queue in the RoutingKey argument).

```
oAMQP->BindQueue("channel_name", "queue_name", "exchange_name", "routing_key");
```

Send Messages

Call the method **PublishMessage** to publish a new AMQP message. The method allows you to publish a **String** or **TStream** message.

```
oAMQP->PublishMessage("channel_name", "exchange_name", "routing_key", "Hello from sgcWebSockets!!!");
```

Receive Messages

AMQP allows you to receive the messages in 2 modes:

- **Request by Client:** using the **GetMessage** method. If there aren't messages in the queue, the event **On-AMQPBasicGetEmpty** will be called.
- **Pushed by Server:** using the Consume method.

Request By Client

```
oAMQP->GetMessage("channel_name", "queue_name");

void OnAMQPGetOk(TObject *Sender, const string aChannel,
  const TsgcAMQPFramePayload_Method_BasicGetOk *aGetOk, const TsgcAMQPMessageContent *aContent)
{
  DoLog("#AMQP_basic_GetOk: " + aChannel + " " + IntToStr(aGetOk->MessageCount) + " " + aContent->Body->AsStr
}
```

Pushed By Server

```
oAMQP->Consume("channel_name", "queue_name");

void OnAMQPGetOk(TObject *Sender, const string aChannel,
  const TsgcAMQPFramePayload_Method_BasicGetOk *aGetOk, const TsgcAMQPMessageContent *aContent)
{
  DoLog("#AMQP_basic_GetOk: " + aChannel + " " + IntToStr(aGetOk->MessageCount) + " " + aContent->Body->AsStr
}
```

Reference

- Properties — 6 properties available on the component.
- Methods — 41 public methods.
- Events — 25 events you can handle.

Connection | Client AMQP Connect

In order to connect to an AMQP Server, you must create first a TsgcWebSocketClient and a TsgcWSPClient_AMQP. Then you must attach AMQP Component to WebSocket Client.

Basic Usage

Connect to AMQP server without authentication. Define the AMQPOptions property values, virtual host and then set in the TsgcWebSocketClient the Host and Port of the server.
If you are using a TCP Plain connection, set the TsgcWebSocketClient property Specifications.RFC6455 to false.

```
oAMQP = new TsgcWSPClient_AMQP();
oAMQP->AMQPOptions->Locale = "en_US";
oAMQP->AMQPOptions->MaxChannels = 100;
oAMQP->AMQPOptions->MaxFrameSize = 16384;
oAMQP->AMQPOptions->VirtualHost = "/";
oAMQP->HeartBeat->Enabled = true;
oAMQP->HeartBeat->Interval = 60;

oClient = new TsgcWebSocketClient();
oAMQP->Client = oClient;
oClient->Specifications->RFC6455 = false;
oClient->Host = "www.esegece.com";
oClient->Port = 5672;
oClient->Active = true;
```

Authentication

If the server requires authentication, use the event **OnAMQPAuthentication** to select the Authentication mechanism (if required) and set the User / Password.

```
oAMQP = new TsgcWSPClient_AMQP();
oAMQP->AMQPOptions->Locale = "en_US";
oAMQP->AMQPOptions->MaxChannels = 100;
oAMQP->AMQPOptions->MaxFrameSize = 16384;
oAMQP->AMQPOptions->VirtualHost = "/";
oAMQP->HeartBeat->Enabled = true;
oAMQP->HeartBeat->Interval = 60;

oClient = new TsgcWebSocketClient();
oAMQP->Client = oClient;
oClient->Specifications->RFC6455 = false;
oClient->Host = "www.esegece.com";
oClient->Port = 5672;
oClient->Active = true;

void OnAMQPAuthentication(TObject *Sender, TsgcAMQPAuthentications *aMechanisms, TsgcAMQPAuthentication &Mechanis
  String &User, String &Password)
{
  User = "user_value";
  Password = "password_value";
}
```

Connection | Client AMQP Disconnect

The client can disconnect a current active connection, using the following methods:

Sending a Close Reason

The AMQP client can inform the server that the connection will be closed and provide information about the reason why it is closing the connection. Use the method Close to request a connection close to the server.

```
oAMQP.Close(541, "Internal Error");
```

Closing Socket Connection

Just set the property Active of TsgcWebSocketClient to False. You can read more about closing connections.

Commands | AMQP Channels

AMQP is a multi-channelled protocol. Channels provide a way to multiplex a heavyweight TCP/IP connection into several light weight connections. This makes the protocol more “firewall friendly” since port usage is predictable. It also means that traffic shaping and other network QoS features can be easily employed.

Every channel runs in its own thread, so every time a new message is received, first the client identifies the channel and queues the message in a queue which is processed by the channel thread.

The channel life-cycle is this:

1. The client opens a new channel (Open).
2. The server confirms that the new channel is ready (Open-Ok).
3. The client and server use the channel as desired.
4. One peer (client or server) closes the channel (Close).
5. The other peer hand-shakes the channel close (Close-Ok).

Open Channel

To create a new channel just call the method **OpenChannel** and pass the channel name as argument. The event **OnAMQPChannelOpen** is raised as a confirmation sent by the server that the channel has been opened.

```
AMQP->OpenChannel("channel_name");

private void OnAMQPChannelOpen(TObject *Sender, const string aChannel)
{
  DoLog("#AMQP_channel_open: " + aChannel);
}
```

A synchronous call can also be done by calling the method **OpenChannelEx**, this method returns true if the channel has been opened and false if no confirmation from server has arrived.

```
if AMQP->OpenChannelEx("channel_name")
{
  DoLog("#AMQP_channel_open channel_name");
}
else
{
  DoLog("#AMQP_channel_open_error");
}
```

Close Channel

To close an existing channel, call the method **CloseChannel** and pass the channel name as argument. The event **OnAMQPChannelClose** will be called when the client receives a confirmation that the channel has been closed.

A Synchronous call can be done calling the method **CloseChannelEx**, this method returns true if the channel has been closed and false if no confirmation from server has arrived.

Channel Flow

Flow control is an emergency procedure used to halt the flow of messages from a peer. It works in the same way between client and server and is implemented by the **EnableChannel / DisableChannel** commands. Flow control is the only mechanism that can stop an over-producing publisher.

To Disable the Flow of a channel, call the method **DisableChannel**, the event **OnAMQPChannelFlow** will be called when the client receives a confirmation that the channel flow has been disabled.
The same applies when enabling the flow of a channel, call the method **EnableChannel**, the event **OnAMQPChannelFlow** will be called when the client receives a confirmation that the channel flow has been enabled.

Synchronous requests are available through the functions **EnableChannelEx** and **DisableChannelEx**.

Commands | AMQP Exchanges

The exchange class lets an application manage exchanges on the server. This class lets the application script its own wiring (rather than relying on some configuration interface). Note: Most applications do not need this level of sophistication, and legacy middleware is unlikely to be able to support this semantic.

The exchange life-cycle is:

1. The client asks the server to make sure the exchange exists (Declare). The client can refine this into, "create the exchange if it does not exist", or "warn me but do not create it, if it does not exist".
2. The client publishes messages to the exchange.
3. The client may choose to delete the exchange (Delete).

Declare Exchange

This method creates a new exchange or verifies that an Exchange already exists. The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **ExchangeName:** it's the name of the exchange, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **ExchangeType:** it's the exchange type, all AMQP servers support "direct" and "fanout" exchanges. Check the server documentation to know which exchanges types are supported.
- **Passive:** if passive is true, the server only verifies that the exchange is already declared. If passive is false, and the exchange does not exist, the server will create a new one.
- **Durable:** if true, the exchange will be re-created when the server starts. If false, the exchange will be deleted when the server stops.
- **AutoDelete:** if true, the exchange will be deleted when all queues have been unbound.
- **Internal:** always false.
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.
- **Arguments:** string which contains custom arguments, the values must be passed as a json string, example: {"x-dead-letter-exchange":"my-dlx"}.

To Declare a new Exchange just call the method **DeclareExchange** and pass the channel name, exchange name and exchange type as arguments. The event **OnAMQPExchangeDeclare** is raised as a confirmation sent by the server that the exchange has been declared.

```
AMQP->DeclareExchange("channel_name", "exchange_name", "direct");

private void OnAMQPExchangeDeclare(TObject *Sender, const string aChannel, const string aExchange)
{
  DoLog("#AMQP_exchange_declare: [" + aChannel + "] " + aExchange);
}
```

A Synchronous call can be done too calling the method **DeclareExchangeEx**, this method returns true if the Exchange has been Declared and false if no confirmation from server has arrived.

```
if AMQP->DeclareExchangeEx("channel_name", "exchange_name", "direct")
{
  DoLog("#AMQP_exchange_declare: [" + aChannel + "] " + aExchange);
}
else
{
  DoLog("#AMQP_exchange_declare_error");
}
```

Delete Exchange

This method is used to delete an existing Exchange. The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **ExchangeName:** it's the name of the exchange, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **IfUnused:** the server only deletes the exchange if there aren't any queues bound to it.
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.

To Delete an existing Exchange call the method **DeleteExchange** and pass the channel name and exchange name as arguments. The event **OnAMQPExchangeDelete** is raised as a confirmation sent by the server that the exchange has been deleted.

A Synchronous call can be done too calling the method **DeleteExchangeEx**, this method returns true if the Exchange has been Deleted and false if no confirmation from server has arrived.

Commands | AMQP Queues

The queue class lets an application manage message queues on the server. This is a basic step in almost all applications that consume messages, at least to verify that an expected message queue is actually present.

The life-cycle for a durable message queue is fairly simple:

1. The client asserts that the message queue exists (Declare, with the "passive" argument).
2. The server confirms that the message queue exists (Declare-Ok).
3. The client reads messages off the message queue.

The life-cycle for a temporary message queue is more interesting:

1. The client creates the message queue (Declare, often with no message queue name so the server will assign a name). The server confirms (Declare-Ok).
2. The client starts a consumer on the message queue. The precise functionality of a consumer is defined by the Basic class.
3. The client cancels the consumer, either explicitly or by closing the channel and/or connection.
4. When the last consumer disappears from the message queue, and after a polite time-out, the server deletes the message queue.

AMQP implements the delivery mechanism for topic subscriptions as message queues. This enables interesting structures where a subscription can be load balanced among a pool of cooperating subscriber applications.

The life-cycle for a subscription involves an extra bind stage:

1. The client creates the message queue (Declare), and the server confirms (Declare-Ok).
2. The client binds the message queue to a topic exchange (Bind) and the server confirms (Bind-Ok).
3. The client uses the message queue as in the previous examples.

Declare Queue

This method creates a new queue or verifies that a Queue already exists. The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **Passive:** if passive is true, the server only verifies that the queue is already declared. If passive is false, and the queue does not exist, the server will create a new one.
- **Durable:** if true, the queue will be re-created when the server starts. If false, the queue will be deleted when the server stops.
- **Exclusive:** if true means the queue is only accessed by the current connection.
- **AutoDelete:** if true, the queue will be deleted when all consumers no longer use the queue.
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.
- **Arguments:** string which contains custom arguments, the values must be passed as a json string, example: {"x-dead-letter-exchange":"my-dlx"}.

To Declare a new Queue just call the method **DeclareQueue** and pass the channel name and queue name as arguments. The event **OnAMQPQueueDeclare** is raised as a confirmation sent by the server that the exchange has been declared.

```
AMQP->DeclareQueue("channel_name", "queue_name");

private void OnAMQPExchangeDeclare(TObject *Sender, const string aChannel, const string aQueue,
  int aMessageCount, int aConsumerCount)
{
  DoLog("#AMQP_queue_declare: [" + aChannel + "] " + aQueue));
}
```

A Synchronous call can be done too calling the method **DeclareQueueEx**, this method returns true if the Queue has been Declared and false if no confirmation from server has arrived.

```
if AMQP->DeclareQueueEx("channel_name", "queue_name")
{
  DoLog("#AMQP_queue_declare: [" + aChannel + "] " + aQueue);
}
else
{
  DoLog("#AMQP_queue_declare_error");
}
```

Delete Queue

This method is used to delete an existing Queue. The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **IfUnused:** the server only deletes the queue if there aren't any consumers attached to it.
- **IfEmpty:** the server only deletes the queue if there are no messages.
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.

To Delete an existing Queue call the method **DeleteQueue** and pass the channel name and queue name as arguments. The event **OnAMQPQueueDelete** is raised as a confirmation sent by the server that the queue has been deleted.

A Synchronous call can be done too calling the method **DeleteQueueEx**, this method returns true if the Queue has been Deleted and false if no confirmation from server has arrived.

Bind Queue

This method is used to bind a Queue to a Exchange. The Exchanges use the bindings to know which queues will be used to route the messages.

All AMQP Servers bind automatically all the queues to the default exchange (it's a "direct" exchange without name) using the Queue Name as the binding routing key. This allows you to send a message to a specific queue without declaring a binding. Just call the method **PublishMessage**, pass an empty value as Exchange Name and set the **RoutingKey** with the value of the **Queue Name**.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **ExchangeName:** it's the name of the exchange, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **RoutingKey:** it's the binding's routing key.
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.
- **Arguments:** string which contains custom arguments, the values must be passed as a json string, example: {"x-dead-letter-exchange":"my-dlx"}.

To bind a Queue to an Exchange call the method **BindQueue** and pass the channel name, queue name, exchange and routing key as arguments. The event **OnAMQPQueueBind** is raised as a confirmation sent by the server that the queue has been bound.

```
AMQP->BindQueueEx("channel_name", "queue_name", "exchange_name", "routing_key");

private void OnAMQPQueueBind(TObject *Sender, const string aChannel, const string aQueue,
  const string aExchange)
{
```

```
  DoLog("#AMQP_queue_bind: [" + aChannel + "] " + aQueue + " -->-- " + aExchange);
}
```

A Synchronous call can be done too calling the method **BindQueueEx**, this method returns true if the Queue has been Bound and false if no confirmation from server has arrived.

```
if AMQP->BindQueueEx("channel_name", "queue_name", "exchange_name", "routing_key")
{
  DoLog("#AMQP_queue_bind: [" + aChannel + "] " + aQueue + " --><-- " + aExchange);
}
else
{
  DoLog("#AMQP_queue_bind_error");
}
```

UnBind Queue

This method deletes an existing queue binding.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **ExchangeName:** it's the name of the exchange, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **RoutingKey:** it's the binding's routing key.

To UnBind a Queue just call the method **UnBindQueue** and pass the channel name, queue name, exchange and routing key as arguments. The event **OnAMQPQueueUnBind** is raised as a confirmation sent by the server that the queue has been unbound.

A Synchronous call can be done too calling the method **UnBindQueueEx**, this method returns true if the Queue has been Unbound and false if no confirmation from server has arrived.

Purge Queue

This method purges all messages of a queue. All the messages that have been sent but are awaiting acknowledgment are not affected.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.

To Purge a Queue just call the method **PurgeQueue** and pass the channel name and queue name as arguments. The event **OnAMQPQueuePurge** is raised as a confirmation sent by the server that the queue has been Purged.

A Synchronous call can be done too calling the method **PurgeQueueEx**, this method returns true if the Queue has been Purged and false if no confirmation from server has arrived.

Commands | AMQP Publish Messages

Publish Messages

The method PublishMessages is used to send a message to the AMQP server.

AMQP Servers automatically bind the queues to "direct" exchange using the queue name as routing key. This allows you to send a message to a specific queue without the need to declare a binding (just calling PublishMessage method and passing the Exchange argument as empty value and the name of the queue in the RoutingKey argument).

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **ExchangeName:** it's the name of the exchange, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **RoutingKey:** it's the binding's routing key name.
- **Mandatory:** if true and the message cannot be routed to any queue, the message is returned by the server, the event OnAMQPBasicReturn is fired.
- **Immediate:** if true and the message cannot be routed to any queue, the message is returned by the server, the event OnAMQPBasicReturn is fired.

```
AMQP->PublishMessage("channel_name", "exchange_name", "routing_key", "Hello from sgcWebSockets!!!");

private void OnAMQPBasicReturn(TObject *Sender, const string aChannel,
  const TsgcAMQPFramePayload_Method_BasicReturn *aReturn,
  const TsgcAMQPMessageContent *aContent)
{
  DoLog("#AMQP_basic_return: " + aChannel + " " + IntToStr(aReturn->ReplyCode) + " " + aReturn->ReplyText + " " +
}
```

Publish Confirmations

The network can fail while publishing a message, the only way to guarantee that a message isn't lost is by using transactions, then for each message/s **select transaction**, **send the message** and **commit**. The confirmation of a successful transaction is received when the event **OnAMQPTransactionOk** is fired.

AMQP Consume Messages

Consumers consume from queues. In order to consume messages there has to be a queue. When a new consumer is added, assuming there are already messages ready in the queue, deliveries will start immediately. The target queue can be empty at the time of consumer registration. In that case first deliveries will happen when new messages are enqueued.

Consuming messages is an **asynchronous** task, which means that every time a new message can be delivered to the consumer queue, it's pushed by the server to the client automatically. You can read an alternative method to Receive Message Synchronously.

Consume

The method **Consume** creates a new consumer in the queue, and every time there is a new message this will be delivered automatically to the consumer client.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer of 255 characters and not begin with "amq." (except if passive parameter is true).
- **ConsumerTag:** it's the name of the consumer and must be unique. If it's not set, then the server creates a new one.
- **NoLocal:** if true means the consumer never consumes messages published on the same channel.
- **NoAck:** if true means the server doesn't expect an acknowledgment for every message delivered.
- **Exclusive:** if true prevents that other consumers consume messages from this queue.
- **NoWait:** if true, the server won't send an acknowledgment to the client.
- **Arguments:** string which contains custom arguments, the values must be passed as a json string, example: {"x-dead-letter-exchange":"my-dlx"}.

The messages are delivered **OnAMQPBasicDeliver** event.

```
AMQP->Consume("channel_name", "queue_name", "consumer_tag");

void OnAMQPBasicDeliver(TObject *Sender, const string aChannel,
  const TsgcAMQPFramePayload_Method_BasicDeliver *aDeliver,
  const TsgcAMQPMessageContent *aContent)
{
  DoLog("#AMQP_basic_deliver: " + aChannel + " " + aDeliver->ConsumerTag + " " +
    " " + aContent->Body->AsString);
}
```

A Synchronous call can be done just calling the method **ConsumeEx**, this method returns true if the Consumer has been created and false if no confirmation from server has arrived.

Cancel Consume

This method is used to Cancel an existing consumer queue.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **ConsumerTag:** it's the name of the consumer.
- **NoWait:** if true, the server won't send an acknowledgment to the client.

```
AMQP->CancelConsume("channel_name", "consumer_tag");
```

```
private void OnAMQPBasicCancelConsume(TObject *Sender, const string aChannel, const string aConsumerTag)
{
  DoLog("#AMQP_basic_cancel_consume: " + aChannel + " " + aConsumerTag);
}
```

A Synchronous call can be done just calling the method **CancelConsumeEx**, this method returns true if the Consumer has been cancelled and false if no confirmation from server has arrived.

Commands | AMQP Get Messages

Getting messages is a **Synchronous** task, which means that it is the client that asks the server if there are messages in the queue. You can read an alternative method to Receive Message Asynchronously.

Get Message

The method **GetMessage** sends a request to the AMQP server asking if there are messages available in a queue. If there are messages these will be dispatched **OnAMQPBasicGetOk** event and if the queue is empty, the event **OnAMQPBasicGetEmpty** will be called.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **NoWait:** if true, the server won't send an acknowledgment to the client.

```
AMQP->GetMessage("channel_name", "queue_name");

private void OnAMQPBasicGetOk(TObject *Sender, const string aChannel,
  const TsgcAMQPFramePayload_Method_BasicGetOk *aGetOk,
  const TsgcAMQPMessageContent *aContent)
{
  DoLog("#AMQP_basic_GetOk: " + aChannel + " " + IntToStr(aGetOk->MessageCount) + " " + aContent->Body->AsString)
}

private void OnAMQPBasicGetEmpty(TObject *Sender, const string aChannel)
{
  DoLog("#AMQP_basic_GetEmpty: " + aChannel);
}
```

A Synchronous call can be done just calling the method **GetMessageEx**, this method returns true if the queue has messages available, otherwise the result will be false.

Commands | AMQP QoS

AMQP allows you to set a QoS level to limit the number of messages the server sends to the client before wait to get the acknowledgment of the messages.

Set QoS

The method **SetQoS** is used to limit the number messages the server sends to the AMQP client.
The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **PrefetchSize:** it's the windows size in bytes, the server doesn't send messages to the client if the total size of all currently unacknowledged messages already sent plus the next message to be sent it's greater than **PrefetchSize** argument. If the value is zero, means no limit.
- **PrefetchCount:** is the maximum number of unacknowledged messages already sent and not acknowledged, if the number is greater, the server stops sending messages to the client.
- **Global:** if true the QoS applies to all existing and new consumers of the connection. If false, the QoS applies to all existing and new consumers of the channel.

The response from the server is received **OnAMQPBasicQoS** event.

```
AMQP->SetQoS("channel_name", 1024000, 100, false);

private void OnAMQPBasicQoS(TObject *Sender, const string aChannel,
  const TsgcAMQPFramePayload_Method_BasicQoS *aQoS)
{
  DoLog("#AMQP_basic_qos: " + aChannel + " " + IntToStr(aQoS->PrefetchSize) + " "
    + IntToStr(aQoS->PrefetchCount) + " " + BoolToStr(aQoS->Global));
}
```

A Synchronous call can be done just calling the method **SetQoSEx**, this method returns true if the request has been processed, otherwise the result will be false.

Commands | AMQP Transactions

AMQP supports two kinds of transactions:

1. Automatic transactions, in which every published message and acknowledgement is processed as a stand-alone transaction.
2. Server local transactions, in which the server will buffer published messages and acknowledgements and commit them on demand from the client.

The Transaction class (“tx”) gives applications access to the second type, namely server transactions. The semantics of this class are:

1. The application asks for server transactions in each channel where it wants these transactions (Select).
2. The application does work (Publish, Ack).
3. The application commits or rolls-back the work (Commit, Roll-back).
4. The application does work, ad infinitum.

Transactions cover published contents and acknowledgements, not deliveries. Thus, a rollback does not requeue or redeliver any messages, and a client is entitled to acknowledge these messages in a following transaction.

The Transaction methods allows publish and ack operations to be batched into atomic units of work. The intention is that all publish and ack requests issued within a transaction will complete successfully or none of them will.

Start Transaction

The method **StartTransaction** starts a new transaction in the server, the client uses this method at least once on a channel before using the Commit or Rollback methods. The event **OnAMQPTransactionOk** is raised when the server acknowledges the use of transactions.

```
AMQP->StartTransaction("channel_name");
```

A Synchronous call can be done just calling the method **StartTransactionEx**, this method returns true if the request has been processed, otherwise the result will be false.

Commit Transaction

This method commits all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a commit. The event **OnAMQPTransactionOk** is raised when the server acknowledges the use of transactions.

```
AMQP->CommitTransaction("channel_name");
```

A Synchronous call can be done just calling the method **CommitTransactionEx**, this method returns true if the request has been processed, otherwise the result will be false.

Rollback Transaction

This method abandons all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a rollback. Note that unacked messages will not be automatically redelivered by rollback; if that is required an explicit recover call should be issued. The event **OnAMQPTransactionOk** is raised when the server acknowledges the use of transactions.

```
AMQP->RollbackTransaction("channel_name");
```

A Synchronous call can be done just calling the method **RollbackTransactionEx**, this method returns true if the request has been processed, otherwise the result will be false.

Protocol Kafka

Apache Kafka is an open-source distributed event-streaming platform used to publish, store and consume streams of records. Producers write messages to topics, the broker stores them in partitioned, append-only logs, and consumers read them back in order, either individually or as part of a coordinated consumer group.
Kafka uses a binary, application-layer protocol over TCP. Each topic is split into one or more partitions, every record in a partition is identified by a monotonically increasing offset, and the broker retains the records so that consumers can read at their own pace and replay history when needed. This makes Kafka well suited to high-throughput messaging, event sourcing, log aggregation and stream processing.

Features

The sgcWebSockets Kafka client talks to the broker directly over the native Kafka protocol, with no REST proxy or external library required.

- Publish (produce) records to any topic and partition, with key, value and configurable acknowledgements.
- Consume records with a managed consumer group or without a group (read all partitions directly).
- Consumer group coordination: coordinator discovery, join/sync, partition assignment and offset commit/fetch.
- Offset management: query the earliest, latest and committed offsets.
- Topic administration: create and delete topics, request cluster metadata and list consumer groups.
- Reads and writes the v2 record batch format used by current Kafka brokers.

Supported Versions

The client uses the v2 record batch format introduced in **Apache Kafka 0.11** and has been tested against **Apache Kafka 3.x**. The default broker port is 9092.

Components

TsgcWSPClient_Kafka: it's the client component that implements the **Apache Kafka** protocol.

Most common uses

- **Connection**
  - Client Kafka Connect
  - Client Kafka Disconnect
- **Commands**
  - Kafka Produce Messages
  - Kafka Consume Messages
  - Kafka Consumer Groups
  - Kafka Topics
  - Kafka Offsets

TsgcWSPClient_Kafka

Apache Kafka client.

Introduction

The **TsgcWSPClient_Kafka** client implements the **Apache Kafka** protocol, talking to the broker directly over the native Kafka binary protocol on a plain TCP connection. It can produce and consume records, manage consumer groups, administer topics and query offsets, with no REST proxy or external library.

Connection

The Kafka client is a protocol component that runs on top of a TsgcWebSocketClient. Because Kafka is not a Web-Socket protocol, set **Specifications.RFC6455** to false so the client connects as a plain TCP socket. Then assign the client to the Kafka component and set the broker **Host** and **Port** (default 9092).
The behaviour of the producer and the consumer is configured through the **KafkaOptions** property:

- **KafkaOptions.ClientId:** identifies your application to the broker.
- **KafkaOptions.Producer.Acks:** required acknowledgements, kafkaAcksNone, kafkaAcksLeader or kafkaAcksAll.
- **KafkaOptions.Producer.TimeoutMs:** how long the broker waits for the required acknowledgements.
- **KafkaOptions.Consumer.GroupId:** the consumer group name. Leave it empty to consume without a group.
- **KafkaOptions.Consumer.OffsetReset:** kafkaOffsetEarliest or kafkaOffsetLatest, where a brand-new group starts reading.

```
TsgcWSPClient_Kafka *oKafka = new TsgcWSPClient_Kafka();
oKafka->KafkaOptions->ClientId = "my-cbuilder-app";
oKafka->OnKafkaConnect = OnKafkaConnect;
oKafka->OnKafkaMessage = OnKafkaMessage;
oKafka->OnKafkaProduce = OnKafkaProduce;

TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
oKafka->Client = oClient;
oClient->Specifications->RFC6455 = false;
oClient->Host = "127.0.0.1";
oClient->Port = 9092;
oClient->Active = true;
```

Produce Messages

Call the method **Produce** to publish a record. The result of every publish is reported through the **OnKafkaProduce** event, including the offset the record was stored at.

```
oKafka->KafkaOptions->Producer->Acks = kafkaAcksLeader;
oKafka->Produce("my-topic", "Hello Kafka", "key-1");
```

Consume Messages

Set a **GroupId** and an **OffsetReset**, subscribe to one or more topics and call **Poll** to fetch the records that are available. Poll returns a TsgcKafkaMessages list and also raises the OnKafkaMessage event for each record. Call Poll repeatedly (for example from a timer) to receive continuously.

```
oKafka->KafkaOptions->Consumer->GroupId = "my-group";
oKafka->KafkaOptions->Consumer->OffsetReset = kafkaOffsetEarliest;
String topics[1] = { "my-topic" };
oKafka->Subscribe(topics, 0);

TsgcKafkaMessages *oMessages = oKafka->Poll(1000);
try {
  for (int i = 0; i < oMessages->Count; i++)
```

```
    DoLog(oMessages->Items[i]->GetValueString());
  } __finally {
    oMessages->Free();
  }
```

Reference

- Properties — 3 properties available on the component.
- Methods — 18 public methods.
- Events — 6 events you can handle.

Connection | Client Kafka Connect

In order to connect to an Apache Kafka broker, you must create first a TsgcWebSocketClient and a TsgcWSPClient_Kafka. Then attach the Kafka Component to the WebSocket Client.
Kafka uses a plain TCP connection, so you must set the TsgcWebSocketClient property Specifications.RFC6455 to false. Set the broker Host and Port (the default Kafka port is 9092), optionally set KafkaOptions.ClientId, and then set Active to true to start the connection.
When the connection to the broker is established, the **OnKafkaConnect** event is raised.

Basic Usage

```
oKafka = new TsgcWSPClient_Kafka();
oKafka->KafkaOptions->ClientId = "sgc-client";

oClient = new TsgcWebSocketClient();
oKafka->Client = oClient;
oClient->Specifications->RFC6455 = false;
oClient->Host = "www.esegece.com";
oClient->Port = 9092;
oClient->Active = true;

void OnKafkaConnect(TObject *Sender)
{
  ShowMessage("Connected to Kafka broker");
}
```

Connection | Client Kafka Disconnect

To disconnect from the Apache Kafka broker, set the WebSocket Client property Active to false. This closes the underlying TCP connection to the broker.
When the connection is closed, the **OnKafkaDisconnect** event is raised.

Basic Usage

```
oClient->Active = false;

void OnKafkaDisconnect(TObject *Sender)
{
  ShowMessage("Disconnected from Kafka broker");
}
```

Commands | Kafka Produce Messages

Use the **Produce(topic, value, key, partition)** method to publish a record to a Kafka topic. The key and partition arguments are optional; when no partition is set, the broker selects one (for example by hashing the key). Call **ProduceBytes** to publish a binary payload.
Configure the delivery guarantee with KafkaOptions.Producer.Acks: kafkaAcksNone (fire and forget), kafkaAcksLeader (wait for the partition leader) or kafkaAcksAll (wait for all in-sync replicas). You can also set Producer.TimeoutMs and Producer.Compression (kafkaCompressionNone or kafkaCompressionGzip).
After a record is sent, the **OnKafkaProduce** event reports the topic, partition, offset and error code of the produced record.

Basic Usage

```
oKafka->KafkaOptions->Producer->Acks = kafkaAcksAll;
oKafka->KafkaOptions->Producer->TimeoutMs = 5000;
oKafka->KafkaOptions->Producer->Compression = kafkaCompressionGzip;

oKafka->Produce("my-topic", "hello world", "key1");

void OnKafkaProduce(TObject *Sender, const String Topic, int Partition, __int64 Offset, int ErrorCode)
{
  if (ErrorCode == 0)
    ShowMessage("Produced to " + Topic + " at offset " + IntToStr(Offset));
  else
    ShowMessage("Produce error " + IntToStr(ErrorCode));
}
```

Commands | Kafka Consume Messages

To consume records, set KafkaOptions.Consumer.GroupId and KafkaOptions.Consumer.OffsetReset (kafkaOffsetEarliest or kafkaOffsetLatest), then call **Subscribe([topic])** with one or more topics.
After subscribing, call **Poll(timeoutMs)** to fetch records. Poll returns a TsgcKafkaMessages list and raises the **OnKafkaMessage** event once per record. You typically call Poll repeatedly, for example from a timer, to keep consuming new records as they arrive.
Each record in the list is a TsgcKafkaMessage. Iterate the list using Count and read each record with GetKeyString, GetValueString, Topic, Partition and Offset.

Subscribe and Poll

```
oKafka->KafkaOptions->Consumer->GroupId = "my-group";
oKafka->KafkaOptions->Consumer->OffsetReset = kafkaOffsetEarliest;
oKafka->Subscribe(ARRAYOFCONST(("my-topic")));

// call repeatedly, e.g. from a TTimer::OnTimer
TsgcKafkaMessages *oMessages = oKafka->Poll(1000);
for (int i = 0; i < oMessages->Count; i++)
  ShowMessage(oMessages->Items[i]->GetKeyString() + " = " + oMessages->Items[i]->GetValueString());
```

OnKafkaMessage Event

Instead of iterating the list returned by Poll, you can handle each record in the OnKafkaMessage event.

```
void OnKafkaMessage(TObject *Sender, TsgcKafkaMessage *Message)
{
  ShowMessage(Message->Topic + ": " + Message->GetKeyString() + " = " + Message->GetValueString());
}
```

Commands | Kafka Consumer Groups

A consumer group lets multiple consumers share the work of reading a topic. The broker tracks the committed offsets per GroupId and assigns the topic partitions to the members of the group, so each partition is read by only one member at a time. Set the group with KafkaOptions.Consumer.GroupId.
For a new group with no committed offsets, KafkaOptions.Consumer.OffsetReset decides where to start: kafkaOffsetEarliest reads from the beginning of the partition, kafkaOffsetLatest reads only new records. Once offsets are committed, consumption resumes from the last committed position.
Call **CommitSync** to commit the offsets of the records returned by the last Poll, so the group resumes after them on the next session. If you leave GroupId empty, the client consumes all partitions directly without joining a group and without committing offsets to the broker.

Basic Usage

```
oKafka->KafkaOptions->Consumer->GroupId = "orders-group";
oKafka->KafkaOptions->Consumer->OffsetReset = kafkaOffsetEarliest;
oKafka->Subscribe(ARRAYOFCONST(("orders")));

TsgcKafkaMessages *oMessages = oKafka->Poll(1000);
if (oMessages->Count > 0)
  oKafka->CommitSync();
```

Commands | Kafka Topics

The Kafka client can administer topics on the broker. Use **CreateTopic(name, partitions, replication)** to create a new topic with the given number of partitions and replication factor, and **DeleteTopic(name)** to remove a topic. Use **GetMetadata([topics])** to retrieve cluster and topic information, such as the brokers, the topics and their partition layout. Pass an empty array to request metadata for the whole cluster.

Basic Usage

```
// create a topic with 3 partitions and replication factor 1
oKafka->CreateTopic("my-topic", 3, 1);

// request metadata for the topic
oKafka->GetMetadata(ARRAYOFCONST(("my-topic")));

// delete the topic
oKafka->DeleteTopic("my-topic");
```

Commands | Kafka Offsets

The Kafka client can query and set offsets for a given topic and partition. **GetEarliestOffset(topic, partition)** returns the oldest available offset, **GetLatestOffset(topic, partition)** returns the next offset to be written, and **GetCommittedOffset(topic, partition)** returns the last offset committed by the consumer group. All three return an Int64.
Use **CommitOffset(topic, partition, offset)** to commit a specific offset for the consumer group, so that consumption resumes from that position on the next session.

Basic Usage

```
__int64 vEarliest = oKafka->GetEarliestOffset("my-topic", 0);
__int64 vLatest = oKafka->GetLatestOffset("my-topic", 0);
__int64 vCommitted = oKafka->GetCommittedOffset("my-topic", 0);

// commit a specific offset for the consumer group
oKafka->CommitOffset("my-topic", 0, vLatest);
```

Protocol AMQP 1.0.0

AMQP (Advanced Message Queuing Protocol) 1.0.0 is a messaging protocol designed for reliable, asynchronous communication between distributed systems. It facilitates the exchange of messages between applications or components in a decoupled manner, allowing them to communicate without direct dependencies. Here's a technical breakdown of some key aspects of AMQP 1.0.0:

- **Message-oriented communication:** AMQP 1.0.0 is centered around the concept of messages. Messages can carry data, instructions, or commands and are the fundamental units of communication.
- **Message Brokers:** The protocol operates on a brokered messaging model. Brokers, which can be servers or intermediary entities, manage the routing and delivery of messages between producers and consumers.
- **Queues and Exchanges:** Queues are storage entities within the broker where messages are temporarily stored. Exchanges define the rules for routing messages from producers to queues based on criteria like message content or routing keys.
- **Addresses and Links:** Addresses identify message destinations within the messaging infrastructure. Links are communication channels between a sender (producer) and a receiver (consumer) associated with a specific address.
- **Sessions and Connections:** Sessions represent a logical channel for communication, allowing multiple streams of messages within a single connection. Connections manage the overall communication link between client applications and the message broker.
- **Security:** AMQP 1.0.0 supports various security mechanisms, including authentication and authorization, to ensure secure communication between clients and brokers.
- **Transport Agnostic:** The protocol is designed to be transport agnostic, meaning it can operate over different network transports such as TCP, TLS, or WebSockets, providing flexibility in deployment.
- **Flow Control:** AMQP 1.0.0 includes mechanisms for flow control, allowing consumers to indicate their ability to handle incoming messages at a given rate. This helps prevent overwhelming consumers with a large number of messages.
- **Error Handling:** The protocol specifies mechanisms for handling errors, including acknowledgment and rejection of messages, ensuring robustness and reliability in message delivery.
- **SASL Authentication:** Simple Authentication and Security Layer (SASL) is used for authenticating and securing connections between clients and brokers.

Overall, AMQP 1.0.0 provides a standardized and interoperable way for different software components and systems to communicate in a loosely coupled manner, making it suitable for various distributed and enterprise-level applications.

Components

TsgcWSPClient_AMQP1: it's the client component that implements **AMQP 1.0.0** protocol.

Most common uses

- **Connection**
  - Client AMQP1 Connect
  - Client AMQP1 Disconnect
  - Client AMQP1 Idle Timeout Connection
  - Client AMQP1 Connection State
  - Client AMQP1 Authentication
  - Client AMQP1 Azure Service Bus

- **Commands**
  - AMQP1 Sessions
  - AMQP1 Links
  - AMQP1 Sender Links
  - AMQP1 Receiver Links
  - AMQP1 Send Message
  - AMQP1 Read Message

TsgcWSPClient_AMQP1

The AMQP 1.

Introduction

The **TsgcWSClient_AMQP1** client implements the **AMQP 1.0.0** protocol following the OASIS specification. The client supports Plain TCP and WebSocket connections, TLS (secure) connections are supported too.

Configuration

The AMQP 1.0.0 client has the property AMQPOptions where you can configure the connection.

- **ChannelMax:** The channel-max value is the highest channel number that can be used on the connection. This
  value plus one is the maximum number of sessions that can be simultaneously active on the connection
- **ContainerId:** (optional) is the name of the source container, identifies uniquely the connection in the server.
- **CreditSize:** default size of the credit flow.
- **IdleTimeout:** The timeout is triggered by a local peer when no frames
  are received after a threshold value is exceeded. The idle timeout is measured in milliseconds, and starts from
  the time the last frame is received.
- **MaxFrameSize:** the max accepted frame size.
- **MaxLinksPerSession:** the max number of links per session.
- **WindowSize:** the default window size.

The AMQP Authentication must be configured in the Authentication property.

- **AuthType:** type of authentication
  - **amqp1authNone:** not configured.
  - **amqp1authSASLAnonymous:** anonymous authentication
  - **amqp1authSASLPlain:** user/password authentication. This type of authentication requires to fill the following properties:
    - Username
    - Password
  - **amqp1authSASLExternal:** external authentication

Connection

The connection starts with the client (usually a messaging application or service) initiating a TCP connection to the server (the message broker). The client connects to the server's port, typically 5672 for non-TLS connections and 5671 for TLS-secured connections. Once the TCP connection is established, the client and server negotiate the AMQP protocol version they will use. AMQP 1.0.0 supports various versions, and during negotiation, both parties agree on using version 1.0.0.

After protocol negotiation, the client may need to authenticate itself to the server, depending on the server's configuration. Authentication mechanisms can include SASL (Simple Authentication and Security Layer) mechanisms like PLAIN, EXTERNAL, or others supported by the server.

Example: connect to AMQP server listening on secure port 5671 and using SASL credentials

```
// Creating AMQP client
oAMQP = new TsgcWSPClient_AMQP1(this);
```

```
// Setting AMQP authentication options
oAMQP->AMQPOptions->Authentication->AuthType = amqp1authSASLPlain;
oAMQP->AMQPOptions->Authentication->Username = L"sgc";
oAMQP->AMQPOptions->Authentication->Password = L"sgc";
// Creating WebSocket client
oClient = new TsgcWebSocketClient(this);
// Setting WebSocket specifications
oClient->Specifications->RFC6455 = false;
// Setting WebSocket client properties
oClient->Host = L"www.esegece.com";
oClient->Port = 5671;
oClient->TLS = true;
// Assigning WebSocket client to AMQP client
oAMQP->Client = oClient;
// Activating WebSocket client
oClient->Active = true;
```

Sessions

Once authenticated, the client opens an AMQP session. A session is a logical context for communication between the client and server. Sessions are used to group related messaging operations together. Use the method **CreateSession** to create a new session, the method allows you to set the session name or leave empty and the component will assign automatically one.

If the session has been created successfully, the event **OnAMQPSessionOpen** will be fired with the details of the session.

```
oAMQP->CreateSession("MySession");
oAMQP->OnAMQPSessionOpen = AMQP1AMQPSessionOpen;
void __fastcall TMyClass::AMQP1AMQPSessionOpen(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Fr
{
    ShowMessage("#session-open: " + aSession->Id);
}
```

Links

Within a session, the client creates links to communicate with specific entities like queues, topics, or other resources provided by the server. Links are bidirectional communication channels used for sending and receiving messages.

The component can work as a sender and receiver node. Allows to create any number of links for each session, up to the limit set in the **MaxLinksPerSession** property.

Sender Links

To create a new sender link, use the method **CreateSenderLink** and pass the name of the session and optionally the name of the sender link. If the link is created successfully, the event **OnAMQPLinkOpen** is raised.

```
oAMQP->CreateSenderLink("MySession", "MySenderLink");
oAMQP->OnAMQPLinkOpen = AMQP1AMQPLinkOpen;
void __fastcall TMyForm::AMQP1AMQPLinkOpen(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *
{
    ShowMessage("#link-open: " + aLink->Name);
}
```

Receiver Links

To create a new receiver link, use the method **CreateReceiverLink** and pass the name of the session and optionally the name of the receiver link. If the link is created successfully, the event **OnAMQPLinkOpen** is raised.

```
oAMQP->CreateSenderLink("MySession", "MySenderLink");
oAMQP->OnAMQPLinkOpen = AMQP1AMQPLinkOpen;
void __fastcall TMyForm::AMQP1AMQPLinkOpen(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *
{
    ShowMessage("#link-open: " + aLink->Name);
}
```

Sending Messages

With the session established and links created, the client can start performing message operations such as sending messages to a destination. Use the method SendMessage to send a message using a sender link.

```
oAMQP->SendMessage("MySession", "MySenderLink", "My first AMQP Message");
```

Receiving Messages

By default, the Receiver Links are created in **Automatic mode**, which means that every time a new message arrives, it will be delivered to the client.

If the Receiver Links has been created in **manual mode**, use the Sync Method **GetMessage** to fetch and wait till a new message arrives.

In Automatic and Manual mode, every time a new message arrives, the event **OnAMQPMessage** is fired.

```
void __fastcall OnAMQPMessageEvent(TObject *Sender, TsgcAMQP1Session *const aSession,
    TsgcAMQP1ReceiverLink *const aLink, TsgcAMQP1Message *const aMessage,
    TsgcAMQP1MessageDeliveryState &DeliveryState)
{
    ShowMessage(aMessage->ApplicationData->AMQPValue->Value);
}
```

Reference

- Properties — 5 properties available on the component.
- Methods — 12 public methods.
- Events — 14 events you can handle.

Connection | Client AMQP1 Connect

In order to connect to an AMQP Server, you must create first a TsgcWebSocketClient and a TsgcWSPClient_AMQP1. Then you must attach AMQP1 Component to WebSocket Client.

After a successful connection, the event OnAMQPConnect is fired.

Basic Usage

Connect to an AMQP 1.0.0 server without authentication. Define the AMQPOptions property values, virtual host and then set in the TsgcWebSocketClient the Host and Port of the server.
If you are using a TCP Plain connection, set the TsgcWebSocketClient property Specifications.RFC6455 to false.

```
// Creating AMQP client
oAMQP = new TsgcWSPClient_AMQP1(this);
// Creating WebSocket client
oClient = new TsgcWebSocketClient(this);
// Setting WebSocket specifications
oClient->Specifications->RFC6455 = false;
// Setting WebSocket client properties
oClient->Host = L"amqp_host_address";
oClient->Port = 5672;
// Assigning WebSocket client to AMQP client
oAMQP->Client = oClient;
// Activating WebSocket client
oClient->Active = true;
```

Authentication

If the server requires authentication, use the properties AMQP:Authentication to set the values of the Username/ Password and set AuthType to the value "amqp1authSASLPlain".

```
// Creating AMQP client
oAMQP = new TsgcWSPClient_AMQP1(this);
// Setting AMQP authentication options
oAMQP->AMQPOptions->Authentication->AuthType = amqp1authSASLPlain;
oAMQP->AMQPOptions->Authentication->Username = L"sgc";
oAMQP->AMQPOptions->Authentication->Password = L"sgc";
// Creating WebSocket client
oClient = new TsgcWebSocketClient(this);
// Setting WebSocket specifications
oClient->Specifications->RFC6455 = false;
// Setting WebSocket client properties
oClient->Host = L"www.esegece.com";
oClient->Port = 5671;
oClient->TLS = true;
// Assigning WebSocket client to AMQP client
oAMQP->Client = oClient;
// Activating WebSocket client
oClient->Active = true;
```

Connection | Client AMQP1 Disconnect

The client can disconnect a current active connection, using the following methods:

Sending a Close Reason

The AMQP client can inform the server that the connection will be closed and provide information about the reason why is closing the connection. Use the method Close to request a connection close to the server.

```
oAMQP.Close('invalid-frame', "The received frame has an invalid format.");
```

Await Close

By default, the **Close** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the Close method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
void Close(const System::UnicodeString aCondition, const System::UnicodeString aDescription)
{
    TsgcAMQP1MethodOptions_Close *oOptions = new TsgcAMQP1MethodOptions_Close();
    try
    {
        oOptions->ErrorCondition = aCondition;
        oOptions->ErrorDescription = aDescription;
        oOptions->Await = true;
        AMQP1->Close(oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

Closing Socket Connection

Just set the property Active of TsgcWebSocketClient to False. You can read more about closing connections.

Connection | Idle Timeout

Connections are subject to an **idle timeout** threshold. The timeout is triggered by the client when no frames are received from the server after a threshold value is exceeded. The idle timeout is measured in milliseconds, and starts from
the time the last frame is received. If the threshold is exceeded the component sends a Close Frame to the server. If the server does not respond after 10 seconds the client will close the TCP socket.

The Value of the Idle Timeout can be configured in the property:

AMQPOptions.IdleTimeout

The value set in this property will be sent to the server when opening the AMQP connection. If the value is greater than zero and less than half the MaxInt value, an internal timer will be enabled to check if the idle timeout has not been exceeded.

Example: set an IdleTimeout value of 60 seconds

AMQPOptions.IdleTimeout = 60000

Connection | Connection State

The AMQP 1.0.0 defines the following connection states:

- **amqp1csUnknown:** initial state.
- **amqp1csStart:** In this state a connection exists, but nothing has been sent or received. This is the state an implementation would be in immediately after performing a socket connect or socket accept.
- **amqp1csHeaderReceived:** In this state the connection header has been received from the peer but a connection header
  has not been sent.
- **amqp1csHeaderSent:** In this state the connection header has been sent to the peer but no connection header has
  been received.
- **amqp1csHeaderExchanged:** In this state the connection header has been sent to the peer and a connection header has
  been received from the peer.
- **amqp1csOpenPipe:** In this state both the connection header and the open frame have been sent but nothing has
  been received.
- **amqp1csOpenClosePipe:** In this state, the connection header, the open frame, any pipelined connection traffic, and
  the close frame have been sent but nothing has been received.
- **amqp1csOpenReceived:** In this state the connection headers have been exchanged. An open frame has been received
  from the peer but an open frame has not been sent.
- **amqp1csOpenSent:** In this state the connection headers have been exchanged. An open frame has been sent
  to the peer but no open frame has yet been received.
- **amqp1csClosePipe:** In this state the connection headers have been exchanged. An open frame, any pipelined
  connection traffic, and the close frame have been sent but no open frame has yet been
  received from the peer.
- **amqp1csOpened:** In this state the connection header and the open frame have been both sent and received.
- **amqp1csCloseReceived:** In this state a close frame has been received indicating that the peer has initiated an AMQP
  close. No further frames are expected to arrive on the connection; however, frames can still
  be sent. If desired, an implementation MAY do a TCP half-close at this point to shut down
  the read side of the connection.
- **amqp1csCloseSent:** In this state a close frame has been sent to the peer. It is illegal to write anything more
  onto the connection, however there could potentially still be incoming frames. If desired,
  an implementation MAY do a TCP half-close at this point to shutdown the write side of the
  connection.
- **amqp1csDiscarding:** The DISCARDING state is a variant of the CLOSE SENT state where the close is triggered
  by an error. In this case any incoming frames on the connection MUST be silently discarded
  until the peer’s close frame is received.
- **amqp1csEnd:** In this state it is illegal for either endpoint to write anything more onto the connection. The connection can be safely closed and discarded.

The AMQP Client has the property **ConnectionState** where you can check in which connection state is the client component.

Connection | AMQP1 Authentication

The component has the following authentication methods:

- **amqp1authNone:** there is no authentication method to use when connecting to the server.
- **amqp1authSASLAnonymous:** connects as anonymous.
- **amqp1authSASLPlain:** the default, uses a user/password authentication.
- **amqp1authSASLExternal:** not currently supported.

SASL Authentication

The most common authentication is using **amqp1authSASLPlain** type. This authentication type, can be enabled in the AMQP1 component, accessing to the property AMQPOptions.Authentication.

- **AuthType:** select amqp1authSASLPlain
- **Username:** the user to use for SASL Authentication.
- **Password:** the secret value to use for SASL Authentication.

The result of the SASL Authentication can be obtained when the event **OnAMQPSASLAuthentication**.

```
void __fastcall OnAMQP1SASLAuthentication(System::TObject* Sender, TsgcAMQP1SaslCode aCode,
    const System::UnicodeString aDescription, bool &Handled)
{
  ShowMessage("#sasl-authentication: " + aDescription);
}
```

Connection | Azure MessageBus

AMQP (Advanced Message Queuing Protocol) is a robust messaging system designed to facilitate communication between diverse containers across various nodes. It standardizes both the protocol for transmitting messages and the structural framework of the messages themselves, ensuring consistent and reliable communication. To dive deeper into the fundamentals of AMQP, refer to our Getting Started with AMQP guide.

The AMQP component within the eSeGeCe library enables seamless integration with leading cloud messaging brokers, including Amazon MQ and Azure Service Bus. This guide focuses on using the AMQP component to connect with Azure Service Bus, demonstrating how to build a multi-tenant application capable of sending and receiving messages efficiently.

The component offers a comprehensive implementation with support for key features such as queues, topics, and subscriptions, making it an ideal choice for modern IoT and enterprise applications.

Azure Configuration

To begin, create a **Service Bus resource** within the Azure Portal. Once the resource is established, make sure to take note of the **resource's domain name**, as it will be essential for integration and configuration.

After the **namespace** has been successfully created, you can manage and monitor it directly from the **namespace overview** in the Azure Portal. This centralized interface provides access to key management tools and settings, enabling seamless administration of your Service Bus resource.

When using **SAS Authentication**, the **username is the SAS Policy name** and the **password is the primary or secondary key**.

```
// Create TCP client
TsgcWebSocketClient* oClient = new TsgcWebSocketClient(nullptr);
oClient->Specifications->RFC6455 = false;
oClient->Host = "esegece.servicebus.windows.net";
oClient->Port = 5671;
oClient->TLS = true;
// Create AMQP1 protocol client
TsgcWSClient_AMQP1* oAMQP1 = new TsgcWSClient_AMQP1(nullptr);
oAMQP1->Specifications->RFC6455 = false;
oAMQP1->AMQPOptions->Authentication->AuthType = amqp1authSASLPlain;
oAMQP1->AMQPOptions->Authentication->Username = "RootManageSharedAccessKey";
oAMQP1->AMQPOptions->Authentication->Password = "BhJ78+w8kMXhS/eE/nBy0cRzodx9tipbi+ASbAXIaH8=";
oAMQP1->Client = oClient;
// Connect to the server
oClient->Active = true;
```

Azure CBS Authentication

Azure Service Bus implements Claims-Based Security (CBS) over AMQP to authorize senders and receivers after the initial SASL handshake. The client opens a management link to the **$cbs** node and sends a **put-token** request containing either a Shared Access Signature (SAS) token or a JSON Web Token (JWT) issued by Microsoft Entra ID. Once the broker validates the token, the authorization is cached for its lifetime and the application can proceed to create sender and receiver links against queues, topics, or subscriptions.

The AMQP1 client automates this flow through two helper methods:

- **CreateAzureCbsSasToken** establishes a CBS sender/receiver link pair, generates a SAS token for the target entity, and publishes it to **$cbs**. Use it when authenticating with a shared access policy.
- **CreateAzureCbsJWT** follows the same CBS exchange but obtains an access token from Microsoft Entra ID (Azure AD) using the client-credentials grant before sending the JWT to **$cbs**.

Both methods require an active AMQP connection and accept the following parameters:

- **aName**: Identifier for the CBS link pair created internally.
- **aNameSpace** and **aEntityName**: The Service Bus namespace (without the *.servicebus.windows.net* suffix) and the queue, topic, or subscription path used to build the token audience.
- **aKeyName** / **aKeyValue**: Shared access policy name and key for SAS tokens. The component signs the token and sends it using the token type *servicebus.windows.net:sastoken*.
- **aTenant**, **aApplicationId**, **aSecret**: Microsoft Entra (Azure AD) directory ID, application (client) ID, and client secret used to request the JWT with the client credentials flow.
- **aListeningPort** (JWT): Local HTTP port for the OAuth 2.0 redirect (defaults to 8080 when not provided).
- **aExpiration** and **aTimeout**: Lifetime of the issued token (in seconds) and the maximum wait time (in milliseconds) for the CBS negotiation.
- **aRaiseIfError**: When set to *True*, the method raises an exception if token acquisition or the CBS response fails.

The following examples illustrate how to authenticate with CBS before sending messages.

```
// Create TCP client
TsgcWebSocketClient* oClient = new TsgcWebSocketClient(nullptr);
oClient->Specifications->RFC6455 = false;
oClient->Host = "esegece.servicebus.windows.net";
oClient->Port = 5671;
oClient->TLS = true;
// Create AMQP1 protocol client
TsgcWSClient_AMQP1* oAMQP1 = new TsgcWSClient_AMQP1(nullptr);
oAMQP1->Specifications->RFC6455 = false;
oAMQP1->AMQPOptions->Authentication->AuthType = amqp1authSASLAnonymous;
oAMQP1->Client = oClient;
// Connect and publish SAS token through CBS
oClient->Active = true;
// wait the client is connected
oAMQP1->CreateAzureCbsSasToken("cbs", "esegece", "queue1",
  "RootManageSharedAccessKey",
  "BhJ78+w8kMXhS/eE/nBy0cRzodx9tipbi+ASbAXIaH8=",
  3600, 10000, true);
```

The next example focuses solely on Microsoft Entra ID (Azure AD) authentication using JWTs. It shows how to request a token with the client credentials flow and publish it to **$cbs** before creating links to send or receive messages.

```
// Create TCP client
TsgcWebSocketClient* oClient = new TsgcWebSocketClient(nullptr);
oClient->Specifications->RFC6455 = false;
oClient->Host = "esegece.servicebus.windows.net";
oClient->Port = 5671;
oClient->TLS = true;
// Create AMQP1 protocol client
TsgcWSClient_AMQP1* oAMQP1 = new TsgcWSClient_AMQP1(nullptr);
oAMQP1->Specifications->RFC6455 = false;
oAMQP1->AMQPOptions->Authentication->AuthType = amqp1authSASLAnonymous;
oAMQP1->Client = oClient;
// Connect and publish JWT through CBS
oClient->Active = true;
oAMQP1->CreateAzureCbsJWT("cbs", "esegece", "queue1",
  "00000000-0000-0000-0000-000000000000", // Tenant ID
  "11111111-1111-1111-1111-111111111111", // Application ID
  "client-secret", 8080, 3600, 10000, true);
```

Commands | AMQP1 Sessions

In the context of the AMQP (Advanced Message Queuing Protocol) 1.0.0 specification, a session represents a logical context for communication between a client and a message broker. Here's a breakdown of what an AMQP 1.0.0 session entails:

- **Logical Context:** A session establishes a logical context for messaging operations between an AMQP client (producer or consumer) and an AMQP broker. It provides a way to group related messaging operations together within a single connection.
- **Communication Channel:** Sessions serve as communication channels over which messages are sent and received. They encapsulate the exchange of messages, acknowledgments, and flow control mechanisms.
- **Transactional Boundaries:** Sessions define transactional boundaries for message operations. They enable the grouping of multiple message sends or receives into a single atomic unit, ensuring that either all operations within the session are processed successfully or none are processed at all.
- **Flow Control:** Sessions support flow control mechanisms to regulate the rate at which messages are exchanged between the client and the broker. Flow control helps prevent overwhelming the resources of either party, ensuring efficient and reliable message delivery.
- **Lifetime Management:** Sessions have a lifecycle that begins when they are created and ends when they are closed. Clients can establish multiple sessions within a single connection to parallelize message processing or isolate message streams.
- **Resource Allocation:** Sessions may be associated with specific resources such as queues, topics, or subscriptions within the broker. Messages sent or received within a session are bound to these resources, enabling targeted message routing and delivery.

In summary, an AMQP 1.0.0 session provides a logical context for message exchange between an AMQP client and broker, facilitating transactional integrity, flow control, and resource management within the messaging system. It defines the boundaries within which messaging operations are performed and helps ensure the efficient and reliable exchange of messages.

Open Session

The method **CreateSession** creates a new session with the given name (or if empty, it creates with a random name), if the session already exists an exception is raised. The client allows you to create multiple session using the same AMQP connection.

Once the session is successfully created, the event **OnAQMPSessionOpen** is fired.

```
oAMQP->CreateSession("MySession");
oAMQP->OnAMQPSessionOpen = OnAMQPSessionOpenEvent;
void __fastcall TMyClass::OnAMQPSessionOpenEvent(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Fram
{
    ShowMessage("#session-open: " + aSession->Id);
}
```

The **CreateSession** method returns the **TsgcAMQP1Session** class which contains the session information.

Await Open Session

By default, the **CreateSession** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CreateSession method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
void OpenSession(const System::UnicodeString aSession)
{
    TsgcAMQP1MethodOptions_SessionOpen *oOptions = new TsgcAMQP1MethodOptions_SessionOpen();
```

```
    try
    {
        oOptions->Await = true;
        AMQP1->CreateSession(aSession, oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

Close Session

To Close an existing session use the method **CloseSession** passing the name of the session to close.

Once the session is successfully closed, the event **OnAQMPSessionClose** is fired.

```
oAMQP->CloseSession("MySession");
oAMQP->OnAMQPSessionClose = OnAMQPSessionCloseEvent;
// OnAMQPSessionCloseEvent implementation
void __fastcall TMyClass::OnAMQPSessionCloseEvent(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Fra
{
    ShowMessage("#session-close: " + aSession->Id + " [" + IntToStr(aSession->Channel) + "] reason: " + aEnd->Err
}
```

Await Close Session

By default, the **CloseSession** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CloseSession method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
void CloseSession(const System::UnicodeString aSession)
{
    TsgcAMQP1MethodOptions_SessionClose *oOptions = new TsgcAMQP1MethodOptions_SessionClose();
    try
    {
        oOptions->Await = true;
        AMQP1->CloseSession(aSession, oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

Commands | AMQP1 Links

In the AMQP (Advanced Message Queuing Protocol) 1.0.0 specification, a link represents a unidirectional communication channel between an AMQP client and a message broker. Let's delve deeper into what AMQP 1.0.0 links entail:

- **Communication Channel:** A link serves as a pathway through which messages flow between an AMQP sender and receiver. It allows for the transmission of messages in one direction, either from the sender to the receiver or vice versa.
- **Unidirectional Flow:** Each link is unidirectional, meaning that messages can only travel in one direction along the link. If bidirectional communication is needed, two links must be established—one for each direction.
- **Message Transfer:** Messages are transferred across links according to the AMQP protocol rules. These messages can include payloads, message properties, and additional metadata required for communication.
- **Resource Binding:** Links are associated with specific resources within the AMQP broker, such as queues, topics, or exchanges. Messages sent or received via a link are directed to or originate from these resources.
- **Flow Control:** Links support flow control mechanisms to regulate the rate at which messages are sent or received. Flow control ensures that neither the sender nor the receiver is overwhelmed by the volume of messages being exchanged.
- **Lifetime Management:** Links have a lifecycle that begins when they are established and ends when they are closed. They can be created dynamically as needed and closed when they are no longer required.
- **Addressing:** Links are identified by unique addresses that specify the source and target endpoints of the communication. These addresses allow clients and brokers to identify and establish connections to the appropriate endpoints.
- **Transactional Boundaries:** Links define transactional boundaries for message operations. They enable the grouping of multiple message sends or receives into a single atomic unit, ensuring consistency and reliability in message delivery.

In summary, AMQP 1.0.0 links provide a means for unidirectional communication between AMQP clients and brokers, facilitating the transfer of messages while supporting flow control, resource binding, addressing, and transactional integrity within the messaging system. They form the fundamental building blocks of message exchange in the AMQP protocol.

There are 2 types of Links:

- **Sender Links:** those links are used to send messages.
- **Receiver Links:** those links are used to receive messages.

Every time a new link is created or deletes, the following events are fired:

- **OnAMQPLinkOpen:** this event is triggered when a new link is created. Use the aLink.Mode property to check if the link is in receiver or sender mode.
- **OnAMQPLinkClose:** this event is triggered when a link is closed.

Commands | AMQP1 Sender Links

In the AMQP 1.0.0 protocol, a **Sender Link** is a **communication channel** established between an AMQP client and an AMQP server for the purpose of **sending messages**. It operates within the context of an AMQP session, which represents a logical channel for communication between the client and server.

Create Sender Link

To **Create a new Sender Link**, call the method **CreateSenderLink** which contains the following parameters:

- **Session:** the session name where the sender link will be attached.
- **Name:** (optional) the name of the sender link, if is not set, a random name will be assigned automatically.
- **Target:** (optional) you can specify the destination where messages should be received on the remote host by setting the "target" parameter. However, in certain scenarios, specifying the target may not be required. In such cases, providing an empty string will be sufficient.
- **SndSettleMode:** (mixed by default) AMQP offers the capability to discuss delivery assurances via the Message Settlement mechanism. Upon establishing a link, both the sender and the receiver discuss and agree upon a settlement mode (one for each role). Senders operate within one of these modes:
  - **amqp1ssmSettled:** The message is considered successfully delivered and acknowledged once it's sent.
  - **amqp1ssmUnsettled:** The message is not considered settled until it's explicitly accepted or rejected by the receiver. This allows for more control over message processing and handling.
  - **amqp1ssmMixed:** A combination of settled and unsettled modes can be used within a single AMQP session. Use the **MessageOptions** parameter of the **SendMessage** method to configure if the message is Settled or not.

When the Sender Link has been created successfully, the event **OnAMQPLinkOpen** will be fired.

```
oAMQP1->CreateSenderLink("MySession", "MySenderLink");
oAMQP1->OnAMQPLinkOpen = AMQP1AMQPLinkOpen;
void __fastcall TMyForm::AMQP1AMQPLinkOpen(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *cons
{
    ShowMessage("#link-open: " + aLink->Name);
}
```

Await Create Sender Link

By default, the **CreateSenderLink** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CreateSenderLink method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
void CreateSenderLink(const System::UnicodeString aSession, const System::UnicodeString aSender)
{
    TsgcAMQP1MethodOptions_CreateSenderLink *oOptions = new TsgcAMQP1MethodOptions_CreateSenderLink();
    try
    {
        oOptions->Await = true;
        AMQP1->CreateSenderLink(aSession, aSender, L"", oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

Sending Messages

To Send a new Message, call the method SendMessage which contains the following parameters:

- **Session:** name of the session.
- **Link:** name of the sender link.
- **Text:** the text of the string message.

```
oAMQP1->SendMessage("MySession", "MySenderLink", "My first AMQP Message");
```

Sending Messages Mixed Mode

When the Sender Link is created in Mixed mode (the default), when sending a message, the user can set if want the message is **settled** or **not**. Use the **MessageOptions** parameter to define if the message is settled or not.

```
TsgcAMQP1MessageOptions* oMessageOptions = new TsgcAMQP1MessageOptions();
try
{
    oMessageOptions->Settled = true;
    oAMQP1->SendMessage("MySession", "MySenderLink", "MyMessage", "message-id", oMessageOptions);
}
__finally
{
    delete oMessageOptions;
}
```

Close Sender Link

To Close an existing Sender Link, call the method **CloseLink** which contains the following parameters:

- Session: name of the session that contains the link.
- Link: name of the sender link.
- Error: (optional) here you can set the reason why the link is closed.

When the Sender Link has been closed successfully, the event **OnAMQPLinkClose** will be fired.

```
oAMQP->CloseLink("MySession", "MySenderLink");
void __fastcall TForm1::OnAMQPLinkCloseEvent(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *cc
{
    ShowMessage("#link-close: " + aLink->Name);
}
```

Await Close Sender Link

By default, the **CloseLink** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CloseLink method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
void CloseSenderLink(const System::UnicodeString aSession, const System::UnicodeString aSenderLink)
{
    TsgcAMQP1MethodOptions_CloseLink *oOptions = new TsgcAMQP1MethodOptions_CloseLink();
    try
    {
        oOptions->Await = true;
        AMQP1->CloseLink(aSession, aSenderLink, oOptions);
    }
    __finally
    {
        delete oOptions;
```

```
    }
}
```

Commands | AMQP1 Receiver Links

In the AMQP 1.0.0 protocol, a **Receiver Link** is a **communication channel** established between an AMQP client and an AMQP server for the purpose of **receiving messages**. It operates within the context of an AMQP session, which represents a logical channel for communication between the client and server.

Create Receiver Link

To **Create a new Receiver Link**, call the method **CreateReceiverLink** which contains the following parameters:

- **Session:** the session name where the sender link will be attached.
- **Name:** (optional) the name of the sender link, if is not set, a random name will be assigned automatically.
- **Source:** (optional) the source can be configured to indicate the location of the node on the remote host that is supposed to act as the sender. In some situations, specifying this address may not be required. In such cases, simply providing an empty string as the value for the parameters will be enough.
- **ReadMode:** (amqp1srmAuto by default) Receiver links can function in one of two modes for receiving messages:
  - **amqp1srmAuto:** Automatic Mode, in this mode the receiver actively works to ensure that messages are received promptly as soon as they become available. It automatically listens for and receives messages without any explicit instruction each time a new message arrives.
  - **amqp1srmManual:** Fetch-Based Mode, in this mode, the receiver will only retrieve or fetch a new message when it is specifically told to do so. Unlike the automatic mode, the receiver will not actively listen for new messages but will instead wait for manual instructions to fetch the next message.
- **RcvSettleMode:** (amqp1rsmFirst by default) Receiver Links operate within one of these modes:
  - **amqp1rsmFirst:** When messages arrive, they will be processed and confirmed right away. If the message hasn't already been confirmed by the time it was sent, the sender will be informed that the message has been received.
  - **amqp1rsmSecond:** Messages that arrive will only be confirmed after the sender has confirmed them first. Additionally, the sender will receive a notification when a message has been received, provided the message wasn't already confirmed when it was sent.

When the Receiver Link has been created successfully, the event **OnAMQPLinkOpen** will be fired.

```
oAMQP1->CreateReceiverLink("MySession", "MyReceiverLink");
oAMQP1->OnAMQPLinkOpen = AMQP1AMQPLinkOpen;
void __fastcall TMyForm::AMQP1AMQPLinkOpen(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *cons
{
    ShowMessage("#link-open: " + aLink->Name);
}
```

Await Create Receiver Link

By default, the **CreateReceiverLink** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CreateReceiverLink method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
void CreateReceiverLink(const System::UnicodeString aSession, const System::UnicodeString aReceiver)
{
    TsgcAMQP1MethodOptions_CreateReceiverLink *oOptions = new TsgcAMQP1MethodOptions_CreateReceiverLink();
    try
    {
        oOptions->Await = true;
        AMQP1->CreateReceiverLink(aSession, aReceiver, L"", oOptions);
    }
    __finally
    {
        delete oOptions;
    }
```

```
}
```

Sync Messages

When the Receiver Link works in Manual ReadMode, call the method **GetMessage** to get new messages. This method is synchronous, which means that waits till a timeout is exceeded (by default 10 seconds). When the method is called, the component increases the credit in one unit and waits till a new message arrives or the timeout has been exceeded. If no message arrives, the credit is set to zero again.

The method **GetMessage** has the following parameters:

- **Session:** name of the session that contains the link.
- **Link:** name of the receiver link.
- **Timeout:** (by default 1000 = 10 seconds) the max time the function will wait to get a new message.

Close Receiver Link

To Close an existing Receiver Link, call the method **CloseLink** which contains the following parameters:

- **Session:** name of the session that contains the link.
- **Link:** name of the receiver link.
- **Error:** (optional) here you can set the reason why the link is closed.

When the Receiver Link has been closed successfully, the event **OnAMQPLinkClose** will be fired.

```
oAMQP1->CloseLink("MySession", "MyReceiverLink");
void __fastcall TForm1::OnAMQPLinkCloseEvent(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *cc
{
    ShowMessage("#link-close: " + aLink->Name);
}
```

Await Close Receiver Link

By default, the **CloseLink** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CloseLink method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
void CloseReceiverLink(const System::UnicodeString aSession, const System::UnicodeString aReceiverLink)
{
    TsgcAMQP1MethodOptions_CloseLink *oOptions = new TsgcAMQP1MethodOptions_CloseLink();
    try
    {
        oOptions->Await = true;
        AMQP1->CloseLink(aSession, aReceiverLink, oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

AMQP1 | Send Message

Read first AMQP1 Sender Links to know how to create a Sender Link.

Send Message

Use the method **SendMessage** passing the Session and SenderLink name to send a text message to the AMQP1 Server. The method has the following parameters:

- **Session:** name of the session.
- **Link:** name of the sender link.
- **Text:** text of the message.
- **MessageId:** (optional) the id of the message, it can be used when using unsettled mode, to know if the server has processed the message.
- **Options:** (optional) allows customizing some options when sending the message.
  - **Settled:** when using a sender link in mixed mode, when sending a message the Settled property can be customized.
  - **Await:** if the message is unsettled, and the value is true, the code will wait till the message is processed by the server or the timeout has exceeded.
  - **Timeout:** value in milliseconds if await is true (by default 10000).
  - **RaiseTimeoutException:** if the timeout is exceeded, an exception is raised (by default true).

```
oAMQP1->SendMessage("MySession", "MySenderLink", "My first AMQP Message");
```

Await Send Message

By default, the **SendMessage** method is asynchronous when sending a message unsettled, setting the property **Await** to true, the client will wait till receives a confirmation from the server that the message has been processed.

```
void SendMessageAwait(const System::UnicodeString aSession, const System::UnicodeString aSenderLink, const System
{
    TsgcAMQP1MethodOptions_SendMessageAck *oOptions = new TsgcAMQP1MethodOptions_SendMessageAck();
    try
    {
        oOptions->Settled = false;
        oOptions->Await = true;
        AMQP1->SendMessage(aSession, aSenderLink, aText, L"messsage-id", oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

Events

When sending a message, there are 2 Events that can be used to know when the message is sent and if the message has been processed by the server (when sending unsettled).

- **OnAMQPMessageSent:** this event is called after the message is sent to the server. When calling the method SendMessage, the message is stored in an internal queue and processed by a secondary thread, so after the message is sent, this event is called.

- **OnAMQPMessageSentAck:** this event is called, when the client receives a confirmation that the message has been processed by the AMQP1 Server.

```
void __fastcall OnAMQPMessageSentAck(System::TObject* Sender, TsgcAMQP1Session* const aSession,
    TsgcAMQP1SenderLink* const aLink, const System::UnicodeString aMessageId,
    TsgcAMQP1FrameDeliveryStates aDeliveryState, TsgcAMQP1FrameDisposition aDisposition)
{
    System::UnicodeString vMessageId = aMessageId;
    switch (aDeliveryState)
    {
        case amqp1fdtsAccepted:
            ShowMessage("#msg-accepted: " + vMessageId);
            break;
        case amqp1fdtsRejected:
            ShowMessage("#msg-rejected: " + vMessageId + " " +
                TsgcAMQP1FrameRejected(aDisposition.State)->Error->Condition + " " +
                TsgcAMQP1FrameRejected(aDisposition.State)->Error->Description);
            break;
        case amqp1fdtsReleased:
            ShowMessage("#msg-released: " + vMessageId);
            break;
        case amqp1fdtsModified:
            ShowMessage("#msg-modified: " + vMessageId + " " +
                TsgcAMQP1FrameModified(aDisposition.State)->MessageAnnotations);
            break;
        case amqp1fdtsReceived:
            ShowMessage("#msg-received: " + vMessageId);
            break;
    }
}
```

AMQP1 | Read Message

Every time a new message is received, the event **OnAMQPMessage** is fired.

The TsgcAMQP1Message instance contains the message received. You can access to the text message using the property **aMessage.ApplicationData.AMQPValue.Value**.

To specify the Delivery Outcome, use the **DeliveryState** parameter. By default, all the messages have the accepted state, but you can set one of the following:

- **amqp1mdtsAccepted:** The message has been processed successfully.
- **amqp1mdtsRejected:** The message failed to process successfully. Set the error using the property DeliveryState.Rejected.
- **amqp1mdtsReleased:** The message has not been and won't be processed.
- **amqp1mdtsModified:** Same as amqp1mdtsReleased, but you can add additional data using the property DeliveryState.Modified.

```
void __fastcall OnAMQPMessage(System::TObject* Sender, TsgcAMQP1Session* const aSession,
    TsgcAMQP1ReceiverLink* const aLink, TsgcAMQP1Message* const aMessage,
    TsgcAMQP1MessageDeliveryState &DeliveryState)
{
    if (aMessage->ApplicationData->AMQPValue->Value == "xxx")
    {
        DeliveryState.State = amqp1mdtsRejected;
        DeliveryState.Rejected->Error->Condition = "amqp-error-processing";
        DeliveryState.Rejected->Error->Description = "Value received was not the expected.";
    }
    else
    {
        DeliveryState.State = amqp1mdtsAccepted;
    }
}
```

Protocol STOMP

STOMP is the Simple (or Streaming) Text Orientated Messaging Protocol. STOMP provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers.

Our STOMP client components support following STOMP versions: 1.0, 1.1 and 1.2.

Components

TsgcWSPClient_STOMP: generic STOMP Protocol client, allows you to connect to any STOMP Server.

TsgcWSPClient_STOMP_RabbitMQ: STOMP client for RabbitMQ Broker.

TsgcWSPClient_STOMP_ActiveMQ: STOMP client for ActiveMQ Broker.

TsgcWSPClient_STOMP

This is the Client Protocol STOMP Component.

Introduction

This is the Client Protocol STOMP Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Reference

- Properties — 8 properties available on the component.
- Methods — 11 public methods.
- Events — 6 events you can handle.

TsgcWSPClient_STOMP_RabbitMQ

This is the Client Protocol STOMP Component for RabbitMQ Broker.

Introduction

This is the Client Protocol STOMP Component for RabbitMQ Broker. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Destinations

The STOMP specification does not prescribe what kinds of destinations a broker must support, instead the value of the destination header in SEND and MESSAGE frames is broker-specific. The RabbitMQ STOMP adapter supports a number of different destination types:

- **Topic:** SEND and SUBSCRIBE to transient and durable topics.
- **Queue:** SEND and SUBSCRIBE to queues managed by the STOMP gateway.
- **QueueOutside:** SEND and SUBSCRIBE to queues created outside the STOMP gateway.
- **TemporaryQueue:** create temporary queues (in reply-to headers only).
- **Exchange:** SEND to arbitrary routing keys and SUBSCRIBE to arbitrary binding patterns.

Reference

- Properties — 8 properties available on the component.
- Methods — 25 public methods.
- Events — 6 events you can handle.

TsgcWSPClient_STOMP_ActiveMQ

This is the Client Protocol STOMP Component for ActiveMQ Broker.

Introduction

This is the Client Protocol STOMP Component for ActiveMQ Broker. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Destinations

The STOMP specification does not prescribe what kinds of destinations a broker must support, instead the value of the destination header in SEND and MESSAGE frames is broker-specific. The Active STOMP adapter supports a number of different destination types:

- **Topic:** SEND and SUBSCRIBE to transient and durable topics.
- **Queue:** SEND and SUBSCRIBE to queues managed by the STOMP gateway.

Publish Options

Note that STOMP is designed to be as simple as possible - so any scripting language/platform can message any other with minimal effort. STOMP allows pluggable headers on each request such as sending & receiving messages. ActiveMQ has several extensions to the Stomp protocol, so that JMS semantics can be supported by Stomp clients. An OpenWire JMS producer can send messages to a Stomp consumer, and a Stomp producer can send messages to an OpenWire JMS consumer. And Stomp to Stomp configurations, can use the richer JMS message control.

STOMP supports the following standard JMS properties on SENT messages:

- **CorrelationId:** Good consumers will add this header to any responses they send.
- **Expires:** Expiration time of the message.
- **JMSXGroupID:** Specifies the Message Groups.
- **JMSXGroupSeq:** Optional header that specifies the sequence number in the Message Groups.
- **Persistent:** Whether or not the message is persistent.
- **Priority:** Priority on the message.
- **ReplyTo:** Destination you should send replies to.
- **MsgType:** Type of the message.

Reference

- Properties — 9 properties available on the component.
- Methods — 16 public methods.
- Events — 6 events you can handle.

Protocol WebRTC

WebRTC (Web Real-Time Communication) is an API definition being drafted by the World Wide Web Consortium (W3C) to enable browser-to-browser applications for voice calling, video chat and P2P file sharing without plugins. The RTC in WebRTC stands for Real-Time Communications, a technology that enables audio/video streaming and data sharing between browser clients (peers). As a set of standards, WebRTC provides any browser with the ability to share application data and perform teleconferencing peer to peer, without the need to install plug-ins or third-party software.

WebRTC components are accessed with JavaScript APIs. Currently, in development are the Network Stream API, which represents an audio or video data stream, and the PeerConnection API, which allows two or more users to communicate browser-to-browser. Also under development is a DataChannel API that enables communication of other types of data for real-time gaming, text chat, file transfer, and so forth.

Components

TsgcWSPServer_WebRTC: Server Protocol WebRTC VCL Component.

Parameters

- **IceServers:** here you can configure turn/stun servers for WebRTC connections. By default uses the following public STUN servers

```
{"iceServers": [{"url": "stun:stun.l.google.com:19302"}]}
```

Browser Test

If you want to test this protocol with your favourite Web Browser, please type this url (you need to define your custom host and port)

```
http://host:port/webrtc.esegece.com.html
```

TsgcWSPServer_WebRTC

This is the Server Protocol WebRTC Component.

Introduction

This is the Server Protocol WebRTC Component. You need to drop this component in the form and select a TsgcWebSocketServer Component using Server Property.

WebRTC Protocol requires STUN/TURN server, demos use public STUN/TURN servers for testing purposes. In order to put in a production system, a dedicated STUN/TURN server is required.

Registered users can download compiled binaries of **Coturn server for Windows**. Read more about COTURN STUN/TURN.

Reference

- Properties — 5 properties available on the component.
- Methods — 2 public methods.
- Events — 11 events you can handle.

Protocol WebRTC Javascript

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/webrtc.esegece.com.js"></script>
```

Open Connection

When a WebSocket connection is opened, the browser requests access to the local camera and microphone, you need to allow access.

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/webrtc.esegece.com.js"></script>
<script>
  var socket = new sgcws_webrtc('ws://{%host%}:{%port%}');
</script>
```

Open WebRTC Channel

When a browser has access to local camera and microphone, 'sgcmediastart' event is fired and then you can attempt to connect to another client using webrtc_connect procedure

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/webrtc.esegece.com.js"></script>
<script>
  var socket = new sgcws_webrtc('ws://{%host%}:{%port%}');
  socket.on('sgcmediastart', function(event)
  {
    socket.webrtc_connect('custom channel');
  }
</script>
```

Close WebRTC channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/webrtc.esegece.com.js"></script>
<script>
  socket.webrtc_disconnect('custom channel');
</script>
```

Protocol WAMP

WAMP is an open WebSocket subprotocol that provides two asynchronous messaging patterns: RPC and PubSub.

Technically, WAMP is an officially registered WebSocket subprotocol (runs on top of WebSocket) that uses JSON as message serialization format.

What is RPC?

Remote Procedure Call (RPC) is a messaging pattern involving peers of two roles: client and server.
A server provides methods or procedures to call under well-known endpoints.
A client calls remote methods or procedures by providing the method or procedure endpoint and any arguments for the call.
The server will execute the method or procedure using the supplied arguments to the call and return the result of the call to the client.

What is PubSub?

Publish & Subscribe (PubSub) is a messaging pattern involving peers of three roles: publisher, subscriber and broker.
A publisher sends (publishes) an event by providing a topic (aka channel) as the abstract address, not a specific peer.
A subscriber receives events by first providing topics (aka channels) it is interested in. Subsequently, the subscriber will receive any events published to that topic.
The broker sits between publishers and subscribers and mediates messages publishes to subscribers. A broker will maintain lists of subscribers per topic so it can dispatch newly published events to the appropriate subscribers.
A broker may also dispatch events on its own, for example when the broker also acts as an RPC server and a method executed on the server should trigger a PubSub event.
In summary, PubSub decouples publishers and receivers via an intermediary, the broker.

Components

TsgcWSPServer_WAMP: Server Protocol WAMP VCL Component.

TsgcWSPClient_WAMP: Client Protocol WAMP VCL Component.

Javascript Component: Client Javascript Reference.

Most Common Uses

- **RPC**
  - Simple RPC
  - RPC Progress Results
- **PubSub**
  - Subscribers
  - Publishers

Browser Test

If you want to test this protocol with your favourite Web Browser, please type this URL (you need to define your custom host and port)

```
http://host:port/wamp.esegece.com.html
```

TsgcWSPServer_WAMP

This is the Server Protocol WAMP Component.

Introduction

This is the Server Protocol WAMP Component. You need to drop this component in the form and select a TsgcWebSocketServer Component using Server Property.

Reference

- Properties — 4 properties available on the component.
- Methods — 4 public methods.
- Events — 14 events you can handle.

TsgcWSPClient_WAMP

This is the Client Protocol WAMP Component.

Introduction

This is the Client Protocol WAMP Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Reference

- Properties — 3 properties available on the component.
- Methods — 7 public methods.
- Events — 13 events you can handle.

Protocol WAMP Javascript

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/wamp.esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
</script>
```

Send New Prefix

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.prefix('sgc', 'http://www.esegece.com');
</script>
```

Request RPC (Remote Procedure Call)

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.call('', 'sgc:CallTest', '20')
</script>
```

Subscribe to a TopicURI

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.subscribe('sgc:test')
</script>
```

UnSubscribe from a TopicURI

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.unsubscribe('sgc:test')
</script>
```

Publish message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.publish('sgc:channel', 'Test Message', [], []);
</script>
```

Show Alert with Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcmessage', function(event)
  {
    alert(event.message);
  }
</script>
```

Show Alert OnCallResult or OnCallError

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.on('wampcallresult', function(event)
  {
    alert('call result: ' + event.CallId + ' - ' + event.CallResult);
  }
  socket.on('wampcallprogressresult', function(event)
  {
    alert('call progress result: ' + event.CallId + ' - ' + event.CallResult);
  }
  socket.on('wampcallerror', function(event)
  {
    alert('call error: ' + event.CallId + ' - ' + event.ErrorURI + ' - ' + event.ErrorDesc +
          ' - ' + event.ErrorDetails);
  }
</script>
```

Show Alert OnEvent

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
```

```
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.on('wampevent', function(event)
  {
    alert('call result: ' + event.TopicURI + ' - ' + event.Event);
  }
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error: ' + event.message);
  };
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  socket.close();
</script>
```

Get Connection Status

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  socket.state();
</script>
```

WAMP | Subscribers

A subscriber receives events by first providing topics (aka channels) it is interested in. Subsequently, the subscriber will receive any events published to that topic.
To receive events from a topic, the subscriber first has to subscribe to that topic.

WAMP Client

```
void OnMessageEvent(TsgcWSConnection *Connection, string Text)
{
  ShowMessage(Text);
}

oClient = new TsgcWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP->Client = oClient.
oClientWAMP->OnMessage = OnMessageEvent;
oClient->Active = true;

// Subscribe to topic after successful connect
oClient->Subscribe("myTopic");
```

WAMP Server

```
void OnSubscriptionEvent(TsgcWSConnection *Connection, string Subscription)
{
  ShowMessage("Subscribed: " + Subscription);
}

oServer = new TsgcWebSocketServer();
oServer->Port = 80;
oServerWAMP = new TsgcWSPServer_WAMP();
oServerWAMP->OnSubscription = OnSubscriptionEvent;
oServerWAMP->Server = oServer;
oServerWAMP->Active = true;
```

WAMP | Publishers

A publisher sends (publishes) an event by providing a topic (aka channel) as the abstract address, not a specific peer. Just call Publish method and pass as arguments the name of the topic and the message you want to send. This message will be delivered to all subscribers of this topic. As a note, there is no need to subscribe to a topic to publish messages on that topic.

There is no need to configure anything on server side, because messages are automatically broadcasted to clients when a publish message is received.

WAMP Client

```
oClient = new TsgcWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP->Client = oClient.
oClientWAMP->OnMessage = OnMessageEvent;
oClient->Active = true;

// Publish a message to all subscribers
oClient->Publish("myTopic", "Hello subscribers myTopic");
```

WAMP | Simple RPC

The most common use of the WAMP component is for a client to request a method from the server, and the server sends a response to the client. The client can send only the name of the method and/or can pass some parameters required by server to calculate the result. Server processes requests and if successful sends a response to client with the result. If there is any error, server sends an error response to client.
As you see, there is only One request and One response (successful or not).

Example: server has a method called **GetTime**, so every time a client requests this method, server returns server time.

WAMP Server

```
void OnServerCall(TsgcWSConnection *Connection, const string CallId, const string ProcUri, const string Arguments
{
  if (ProcUri == "GetTime")
  {
    oServerWAMP->CallResult(CallId, FormatDateTime("yyyymmdd hh:nn:ss", Now));
  }
  else
  {
    oServer->WAMP->CallError(CallId, "Unknown method");
  }
}
oServer = new TsgcWebSocketServer();
oServer->Port = 80;
oServerWAMP = new TsgcWSPServer_WAMP();
oServerWAMP->OnCall = OnServerCallEvent();
oServerWAMP->Server = oServer;
oServer->Active = true;
```

WAMP Client

```
void OnCallResultClient(TsgcWSConnection *Connection, string CallId, string Result);
{
  ShowMessage(Result);
}
void OnCallErrorClient(TsgcWSConnection *Connection, string Error)
{
  ShowMessage(Error);
}
oClient = new TsgcWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP->OnCallResult = OnCallResultClient;
oClientWAMP->OnCallError = OnCallErrorClient;
oClientWAMP->Client = oClient;
oClient->Active = true;
// After client has connected, request GetTime from server
oClientWAMP->Call("GetTime");
```

WAMP | RPC Progress Results

Sometimes, Remote Procedure Calls require more than one result to finish requests, by default WAMP 1.0 protocol doesn't allow Partial results in a call, this is a feature only for sgcWebSockets library.
The flow is very similar to a simple RPC, but here there are 1 or more partial results before CallResult is called to finish the process.
Basically, a client requests a procedure from the server, and the server can send a result or an error. If it sends a result, this can be the final result or it must send more results later. If it's final result, will call method **CallResult** and the process will be finished. If there are more results to send, will call method **CallProgressResult**.

Example: client requests server a method to receive every second the server time and stop after 20 messages.

WAMP Server

```
void OnServerCall(TsgcWSConnection *Connection, const string CallId, const string ProcUri, const string Arguments
{
  if (ProcUri == "GetProgressiveTime")
  {
    int vNum = StrToInt(Arguments);
        for (int i = 1; i = vNum; i++)
        {
          if (i == 20)
          {
        oServerWAMP->CallResult(CallId, FormatDateTime("yyyymmdd hh:nn:ss", Now));
          }
          else
          {
                oServerWAMP->CallProgressiveResult(CallId, FormatDateTime("yyyymmdd hh:nn:ss", Now));
          }
        }
  }
  else
  {
    oServer->WAMP->CallError(CallId, "Unknown method");
  }
}


oServer = new TsgcWebSocketServer();
oServer->Port = 80;
oServerWAMP = new TsgcWSPServer_WAMP();
oServerWAMP->OnCall = OnServerCallEvent();
oServerWAMP->Server = oServer;
oServer->Active = true;
```

WAMP Client

```
void OnCallResultClient(TsgcWSConnection *Connection, string CallId, string Result);
{
  ShowMessage(Result);
}


void OnCallProgressResultClient(TsgcWSConnection *Connection, string CallId, string Result);
{
  ShowMessage(Result);
}
void OnCallErrorClient(TsgcWSConnection *Connection, string Error)
{
  ShowMessage(Error);
}


oClient = new TsgcWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP->OnCallResult = OnCallResultClient;
oClientWAMP->OnCallProgressResult = OnCallProgressResultClient;
oClientWAMP->OnCallError = OnCallErrorClient;
oClientWAMP->Client = oClient;
```

```
oClient->Active = true;
// After client has connected, request GetTime from server
oClientWAMP->Call("GetProgressTime");
```

Protocol WAMP 2

WAMP provides Unified Application Routing in an open WebSocket protocol that works with different languages.

Using WAMP you can build distributed systems out of application components which are loosely coupled and communicate in (soft) real-time.

At its core, WAMP offers two communication patterns for application components to talk to each other:

- Publish & Subscribe (PubSub)
- Remote Procedure Calls (RPC)

WAMP is easy to use, simple to implement and based on modern Web standards: WebSocket, JSON and URIs.

Components

TsgcWSPClient_WAMP2: Client Protocol WAMP2 VCL Component.

TsgcWSPClient_WAMP2

This is the Client Protocol WAMP2 Component.

Introduction

This is the Client Protocol WAMP2 Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Session Methods

- **ABORT:** Both the Router and the Client may abort the opening of a WAMP session by sending an ABORT message.

  Reason MUST be an URI.
  Details MUST be a dictionary that allows you to provide additional, optional closing information (see below).

  No response to an ABORT message is expected.

- **GOODBYE:** A WAMP session starts its lifetime with the Router sending a WELCOME message to the Client and ends when the underlying transport disappears or when the WAMP session is closed explicitly by a GOODBYE message sent by one Peer and a GOODBYE message sent from the other Peer in response.

  Reason MUST be a URI.
  Details MUST be a dictionary that allows providing additional, optional closing information.

Publish/Subscribe Methods

- **PUBLISH:** When a Publisher requests to publish an event to some topic, it sends a PUBLISH message to a Broker:

  Request is a random, ephemeral ID chosen by the Publisher and used to correlate the Broker's response with the request.
  Options is a dictionary that allows you to provide additional publication request details in an extensible way. This is described further below.
  Topic is the topic published to.
  Arguments is a list of application-level event payload elements. The list may be of zero length.
  ArgumentsKw is an optional dictionary containing application-level event payload, provided as keyword arguments. The dictionary may be empty.

  If the Broker is able to fulfil and allowing the publication, the Broker will send the event to all current Subscribers of the topic of the published event.
  By default, publications are unacknowledged, and the Broker will not respond, whether the publication was successful indeed or not.

- **SUBSCRIBE:** A Subscriber communicates its interest in a topic to a Broker by sending a SUBSCRIBE message:

  Request MUST be a random, ephemeral ID chosen by the Subscriber and used to correlate the Broker's response with the request.
  Options MUST be a dictionary that allows providing additional subscription request details in an extensible way.
  Topic is the topic the Subscriber wants to subscribe to and MUST be a URI.

- **UNSUBSCRIBE:** When a Subscriber is no longer interested in receiving events for a subscription it sends an UNSUBSCRIBE message

  Request MUST be a random, ephemeral ID chosen by the Subscriber and used to correlate the Broker's response with the request.
  SUBSCRIBED.Subscription MUST be the ID for the subscription to unsubscribe from, originally handed out by the Broker to the Subscriber.

RPC Methods

- **CALL:** When a Caller wishes to call a remote procedure, it sends a CALL message to a Dealer:

  Request is a random, ephemeral ID chosen by the Caller and used to correlate the Dealer's response with the request.
  Options is a dictionary that allows you to provide additional call request details in an extensible way. This is described further below.
  Procedure is the URI of the procedure to be called.
  Arguments is a list of positional call arguments (each of arbitrary type). The list may be of zero length.
  ArgumentsKw is a dictionary of keyword call arguments (each of arbitrary type). The dictionary may be empty.

- **REGISTERCALL:** A Callee announces the availability of an endpoint implementing a procedure with a Dealer by sending a REGISTER message:

  Request is a random, ephemeral ID chosen by the Callee and used to correlate the Dealer's response with the request.
  Options is a dictionary that allows providing additional registration request details in an extensible way. This is described further below.
  Procedure is the procedure the Callee wants to register

- **UNREGISTERCALL:** When a Callee is no longer willing to provide an implementation of the registered procedure, it sends an UNREGISTER message to the Dealer:

  Request is a random, ephemeral ID chosen by the Callee and used to correlate the Dealer's response with the request.
  REGISTERED.Registration is the ID for the registration to revoke, originally handed out by the Dealer to the Callee.

- **INVOCATION:** If the Dealer is able to fulfil (mediate) the call and it allows the call, it sends a INVOCATION message to the respective Callee implementing the procedure:

  Request is a random, ephemeral ID chosen by the Dealer and used to correlate the Callee's response with the request.
  REGISTERED.Registration is the registration ID under which the procedure was registered at the Dealer.
  Details is a dictionary that allows you to provide additional invocation request details in an extensible way. This is described further below.
  CALL.Arguments is the original list of positional call arguments as provided by the Caller.
  CALL.ArgumentsKw is the original dictionary of keyword call arguments as provided by the Caller.

- **YIELD:** If the Callee is able to successfully process and finish the execution of the call, it answers by sending a YIELD message to the Dealer:

  INVOCATION.Request is the ID from the original invocation request.
  Options is a dictionary that allows providing additional options.
  Arguments is a list of positional result elements (each of arbitrary type). The list may be of zero length.
  ArgumentsKw is a dictionary of keyword result elements (each of arbitrary type). The dictionary may be empty.

Reference

- Properties — 3 properties available on the component.
- Methods — 11 public methods.
- Events — 20 events you can handle.

Protocol Default

This is default sub-protocol implemented using "JSONRPC 2.0" messages, every time you send a message using this protocol, a JSON object is created with the following properties:

jsonrpc: A String specifying the version of the JSON-RPC protocol. MUST be exactly "2.0".

method: A String containing the name of the method to be invoked. Method names that begin with the word rpc followed by a period character (U+002E or ASCII 46) are reserved for rpc-internal methods and extensions and MUST NOT be used for anything else.

params: A Structured value that holds the parameter values to be used during the invocation of the method. This member MAY be omitted.

id: An identifier established by the Client that MUST contain a String, Number, or NULL value if included. If it is not included it is assumed to be a notification. The value SHOULD normally not be Null [1] and Numbers SHOULD NOT contain fractional parts [2]

JSON object example:

```
{"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
```

Features

- **Publish/subscribe** message pattern to provide one-to-many message distribution and decoupling of applications. Supports Wildcard characters, so you can subscribe to a hierarchy of channels. Example: if you want to subscribe to all channels which start with 'news', then call Subscribe('news*').
- A messaging transport that is **agnostic** to the content of the payload
- **Acknowledgment** of messages sent.
- Supports **transactional messages** through server local transactions. When the client commits the transaction, the server processes all messages queued. If the client rolls back the transaction, then all messages are deleted.
- Implements **QoS** (Quality of Service) for message delivery.

Components

TsgcWSPClient_sgc: Server Protocol Default VCL Component.

TsgcWSPClient_sgc: Client Protocol Default VCL Component.

Javascript Component: Client Javascript Reference.

Browser Test

If you want to test this protocol with your favourite Web Browser, please type this URL (you need to define your custom host and port)

```
http://host:port/esegece.com.html
```

TsgcWSPServer_sgc

This is the Server Protocol Default Component.

Introduction

This is the Server Protocol Default Component. You need to drop this component in the form and select a TsgcWebSocketServer Component using Server Property.

Reference

- Properties — 6 properties available on the component.
- Methods — 6 public methods.
- Events — 16 events you can handle.

TsgcWSPClient_sgc

This is the Client Protocol Default Component.

Introduction

This is the Client Protocol Default Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Reference

- Properties — 5 properties available on the component.
- Methods — 11 public methods.
- Events — 15 events you can handle.

TsgcIWWSPClient_sgc

This is the Intraweb Client Protocol Default Component.

Introduction

This is the Intraweb Client Protocol Default Component. You need to drop this component in the form and select a TsgcIWWebSocketClient Component using Client Property.

Reference

- Properties — 2 properties available on the component.
- Methods — 3 public methods.
- Events — 6 events you can handle.

Protocol Default Javascript

Default Protocol Javascript sgcWebSockets uses **sgcWebSocket.js and esegece.com.js** files.

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
</script>
```

Send Message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.send('Hello sgcWebSockets!');
</script>
```

Show Alert with Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcmessage', function(event)
  {
    alert(event.message);
  }
</script>
```

Publish Message to test channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.publish('Hello sgcWebSockets!', 'test');
</script>
```

Show Alert with Event Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcevent', function(event)
  {
    alert('channel:' + event.channel + '. message: ' + event.message);
  }
</script>
```

Call RPC

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  var params = {param:10};
  socket.rpc(GUID(), 'test', JSON.stringify(params));
</script>
```

Handle RPC Response

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcrpcresult', function(event)
  {
    alert('result:' + event.result);
  }
  socket.on('sgcrpcerror', function(event)
  {
    alert('error:' + event.code + ' ' + event.message);
  }
</script>
```

Call Notify

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  var params = {param:10};
  socket.notify('test', JSON.stringify(params));
</script>
```

Send Messages in a Transaction

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.starttransaction('sgc:test');
  socket.publish('Message1', 'sgc:test');
  socket.publish('Message2', 'sgc:test');
```

```
  socket.publish('Message3', 'sgc:test');
  socket.commit('sgc:test');
</script>
```

Show Alert OnSubscribe or OnUnSubscribe to a channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcsubscribe', function(event)
  {
    alert('subscribed: ' + event.channel);
  }
  socket.on('sgcunsubscribe', function(event)
  {
    alert('unsubscribed: ' + event.channel);
  }
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error: ' + event.message);
  };
</script>
```

Get Session

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcsession', function(event)
  {
    alert(event.guid);
  };
  socket.getsession();
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  socket.close();
</script>
```

Get Connection Status

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.state();
</script>
```

Set QoS

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.qoslevel1();
  socket.publish('message', 'channel');
</script>
```

Set Queue Level

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.queuelevel2();
  socket.publish('message1', 'channel1');
  socket.publish('message2', 'channel1');
</script>
```

Protocol Dataset

This protocol inherits from Protocol Default and it's useful if you want to broadcast dataset changes over clients connected to this protocol. It can be used in 2 modes:

1. Replicate database: the database changes are replicated to all client databases, example: a server has a database with stock quotes and all connected clients receive quotes changes. There is a single database (in server) and every client has its own database. Every time a quote is updated, this change is broadcasted to all connected clients and every client updates its own record database. Use UpdateMode: upWhereAll or upWhereChanged for this mode type.

2. Database updates: here there is a single database shared by server and clients, and every time there is a client that updates a record in a database, all other clients want to be notified about this update. Use UpdateMode: upRefreshAll for this mode.

Most common uses

- **Update Mode**
  - How Replicate Table
  - How Notify Updates

It uses "JSON-RPC 2.0" Object, and every time there is a dataset change, it sends all field values (* only fields supported) using Dataset Object.

To allow the component to search records on the dataset, you need to specify which fields are the Key, **example:** if in your dataset, ID field is the key you will need to write a code like this

```
void OnAfterOpenDataSet(TDataSet *DataSet)
{
  DataSet->FieldByName("ID")->ProviderFlags =
    Dataset->FieldByName("ID")->ProviderFlags + [pfInKey];
}
```

Components

TsgcWSPServer_Dataset: Server Protocol Dataset VCL Component.

TsgcWSPClient_Dataset: Client Protocol Dataset VCL Component.

Javascript Component: Client Javascript Reference.

Browser Test

If you want to test this protocol with your favourite Web Browser, please type this URL (you need to define your custom host and port)

```
http://host:port/dataset.esegece.com.html
```

TsgcWSPServer_Dataset

This is the Server Protocol Dataset Component.

Introduction

This is the Server Protocol Dataset Component. You need to drop this component in the form and select a TsgcWebSocketServer Component using Server Property and select a Dataset Component using Dataset Property.

This component inherits from TsgcWSProtocol_Server_sgc all methods and properties.

Reference

- Properties — 13 properties available on the component.
- Methods — 9 public methods.
- Events — 22 events you can handle.

TsgcWSPClient_Dataset

This is the Client Protocol Dataset Component.

Introduction

This is the Client Protocol Dataset Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property and select a Dataset Component using Dataset Property.

This component inherits from TsgcWSProtocol_Client_sgc all methods and properties.

Reference

- Properties — 13 properties available on the component.
- Methods — 14 public methods.
- Events — 25 events you can handle.

TsgcIWWSPClient_Dataset

This is the Intraweb Client Protocol Dataset Component.

Introduction

This is the Intraweb Client Protocol Dataset Component. You need to drop this component in the form and select a TsgcIWWebSocketClient Component using Client Property and select a Dataset Component using Dataset Property.

This component inherits from TsgcIWWSPClient_sgc all methods and properties.

Reference

- Properties — 3 properties available on the component.
- Methods — 6 public methods.
- Events — 9 events you can handle.

Protocol Dataset Javascript

Dataset Protocol Javascript sgcWebSockets uses **sgcWebSocket.js and dataset.esegece.com.js** files.

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/dataset.esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws_dataset('ws://{%host%}:{%port%}');
</script>
```

Send Message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws_dataset('ws://{%host%}:{%port%}');
  socket.send('Hello sgcWebSockets!');
</script>
```

Show Alert with Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcdataset', function(event)
  {
    alert(event.dataset);
  }
</script>
```

Show Alert with Dataset Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws_dataset('ws://{%host%}:{%port%}');
  socket.on('sgcmessage', function(event)
  {
    alert(event.message);
```

```
  }
</script>
```

Show Alert OnSubscribe or OnUnSubscribe to a channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws_dataset('ws://{%host%}:{%port%}');
  socket.on('sgcsubscribe', function(event)
  {
    alert('subscribed: ' + event.channel);
  }
  socket.on('sgcunsubscribe', function(event)
  {
    alert('unsubscribed: ' + event.channel);
  }
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws_dataset('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error: ' + event.message);
  };
</script>
```

Subscribe All Dataset Changes

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  socket.subscribe_all();
</script>
```

UnSubscribe All Dataset Changes

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  socket.unsubscribe_all();
</script>
```

Handle Dataset Changes

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
var socket = new sgcws_dataset('ws://{%host%}:{%port%}');

socket.on('sgcdataset', function(evt){

if ((evt.channel == "sgc@dataset@new") || (evt.channel == "sgc@dataset@update")) {

... here you need to implement your own code insert/update records ...
}
else if (evt.channel == "sgc@dataset@delete") {

... here you need to implement your own code to delete records ...

}
});
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  socket.close();
</script>
```

Get Connection Status

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  socket.state();
</script>
```

Protocol Dataset | Replicate Table

This mode tries to solve a common scenario where a **table is replicated** for all connected clients, **example**, if you have a server with a **stock quotes table**, you want **broadcast stock changes** to all **clients**, but you don't want that a client can connect to your database. So, every time there is a **change** in any stock quotes, the **record information will be broadcasted** to all connected clients. Every **client will read the record** and **update** his own table.

You can check in Demos folder, SQLLite/MultipleDatabase demo.

Configure Dataset Server

Create a new Dataset Protocol Server and configure using the following properties

- **ApplyUpdates:** set to **True**, every time there is a change, this will be broadcasted to clients
- **AutoSynchronize:** set to **True**, every time a new client connects to server, server will send all records (metadata and data), so client will get latest information from server.
- **UpdateMode:** set to **upWhereAll** or **upWhereChanged**. The difference is the first sends all fields of a record and the second only fields changed in an update.

```
TsgcWebSocketServer *oServer = new TsgcWebSocketServer();
TsgcWSPServer_Dataset *oProtocolDataset = new TsgcWSPServer_Dataset();
oProtocolDataset->Server = oServer;
oProtocolDataset->Dataset = <...your dataset..>;
oProtocolDataset->ApplyUpdates = true;
oProtocolDataset->AutoSynchronize = true;
oProtocolDataset->NotifyUpdates = true;
oProtocolDataset->UpdateMode = upWhereAll;
oServer->Port = 80;
oServer->Active = true;
```

Configure Dataset Client

Create a new Dataset Protocol Client and configure using the following properties

- **ApplyUpdates:** set to **True**, every time there is a change, this will be sent to server.
- **AutoSubscribe:** set to **True**, every time a new client connects to server, the client subscribes automatically to update, delete and new record.
- **UpdateMode:** set to **upWhereAll** or **upWhereChanged**. The difference is the first sends all fields of a record and the second only fields changed in an update.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSPClient_Dataset *oProtocolDataset = new TsgcWSPClient_Dataset();
oProtocolDataset->Client = oClient;
oProtocolDataset->Dataset = <...your dataset..>;
oProtocolDataset->ApplyUpdates = true;
oProtocolDataset->AutoSubscribe = true;
oProtocolDataset->NotifyUpdates = true;
oProtocolDataset->UpdateMode = upWhereAll;
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClient->Active = true;
```

Protocol Dataset | Notify Updates

This mode tries to solve a scenario where **server** and **clients** share a **single database** (server and clients are connected to the same physical database) and clients want to be notified every time other client has done any change on a dataset.

You can check in Demos folder, SQLLite/SingleDatabase demo.

Configure Dataset Server

Create a new Dataset Protocol Server and configure using the following properties

- **ApplyUpdates:** set to **True**, every time there is a change, this will be broadcasted to clients
- **AutoSynchronize:** set to **False**, here is not needed to set to true, because client is connected to the same database than server.
- **UpdateMode:** set to **upRefreshAll**.

```
TsgcWebSocketServer *oServer = new TsgcWebSocketServer();
TsgcWSPServer_Dataset *oProtocolDataset = new TsgcWSPServer_Dataset();
oProtocolDataset->Server = oServer;
oProtocolDataset->Dataset = <...your dataset..>;
oProtocolDataset->ApplyUpdates = true;
oProtocolDataset->AutoSynchronize = false;
oProtocolDataset->NotifyUpdates = true;
oProtocolDataset->UpdateMode = upRefreshAll;
oServer->Port = 80;
oServer->Active = true;
```

Configure Dataset Client

Create a new Dataset Protocol Client and configure using the following properties

- **ApplyUpdates:** set to **True**, every time there is a change, this will be sent to server.
- **AutoSubscribe:** set to **True**, every time a new client connects to server, the client subscribes automatically to update, delete and new record.
- **UpdateMode:** set to **upRefreshAll**.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSPClient_Dataset *oProtocolDataset = new TsgcWSPClient_Dataset();
oProtocolDataset->Client = oClient;
oProtocolDataset->Dataset = <...your dataset..>;
oProtocolDataset->ApplyUpdates = true;
oProtocolDataset->AutoSubscribe = true;
oProtocolDataset->NotifyUpdates = true;
oProtocolDataset->UpdateMode = upRefreshAll;
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClient->Active = true;
```

Protocol Files

This protocol allows sending files using binary WebSocket transport. It can handle big files with a low memory usage.

Features

- **Publish/subscribe** message pattern to provide one-to-many message distribution and decoupling of applications.
- **Acknowledgment** of messages sent.
- Implements **QoS** (Quality of Service) for file delivery.
- Optionally can request **Authorization** for files received.
- **Low memory** usage.

Components

TsgcWSPServer_Files: Server Protocol Files VCL Component.

TsgcWSPClient_Files: Client Protocol Files VCL Component.

Classes

TsgcWSMessageFile: the object which encapsulates file packet information.

Most common uses

- **Send Files**
  - How Send Files To Server
  - How Send Files To Clients
- **Big Files**
  - How Send Big Files

TsgcWSPServer_Files

This is the Server Files Protocol Component.

Introduction

This is the Server Files Protocol Component. You need to drop this component in the form and select a TsgcWebSocketServer Component using Server Property.

Reference

- Properties — 5 properties available on the component.
- Methods — 3 public methods.
- Events — 20 events you can handle.

TsgcWSPClient_Files

This is the Client Files Protocol Component.

Introduction

This is the Client Files Protocol Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Reference

- Properties — 5 properties available on the component.
- Methods — 4 public methods.
- Events — 19 events you can handle.

TsgcWSMessageFile

This object is passed as a parameter every time a file protocol event is raised.

Properties

- BufferSize: default size of the packet.
- Channel: if specified, this file will only be sent to clients subscribed to specific channel.
- Method: internal method.
- FileId: identifier of a file; it is unique for all files received/sent.
- Data: user custom data. Here the user can set whatever text.
- FileName: name of the file.
- FilePosition: file position in bytes.
- FileSize: Total file size in bytes.
- Id: identifier of a packet; it is unique for every packet.
- QoS: quality of service of the message.
- Streaming: for internal use.
- Text: for internal use.

Protocol Files | How Send Files To Server

To send a file to the server, just call the method **SendFile** of Files Protocol and pass the full **FileName** as argument.
The file received by the server will be saved by default in the same directory where the server executable is located or in the Path set in the Files.SaveDirectory property.

```
// ... Create Server
TsgcWebSocketServer oServer = new TsgcWebSocketServer();
TsgcWSPServer_Files oServer_Files = new TsgcWSPServer_Files();
oServer_Files->Server = oServer;
oServer->Host = "127.0.0.1";
oServer->Port = 8080;


// ... Create Client
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient->URL = "ws://127.0.0.1:8080";


// ... Create Protocol
TsgcWSPClient_Files oClient_Files = new TsgcWSPClient_Files();
oClient_Files->Client = oClient;


// ... Start Server
oServer->Active = true;


// ... Connect client and Send File
if oClient->Connect() then
  oClient_Files->SendFile("c:\Documents\yourfile.txt");
```

Protocol Files | How Send Files To Clients

To send a file to a client, just call the method **SendFile** of Files Protocol and pass the **Guid** of the Connection and the full **FileName** as argument. The Guid of the client connection can be captured OnConnect event of Server Protocol Files.
The file received by the client will be saved by default in the same directory where the client executable is located or in the Path set in the Files.SaveDirectory property.

```
// ... capture the guid of the client connection to send later the file
void OnConnectEvent(TsgcWSConnection *Connection)
{
  FGuid = Connection->Guid;
}


// ... Create Server
TsgcWebSocketServer oServer = new TsgcWebSocketServer();
TsgcWSPServer_Files oServer_Files = new TsgcWSPServer_Files();
oServer_Files->Server = oServer;
oServer_Files->OnConnect = OnConnectEvent;
oServer->Host = "127.0.0.1";
oServer->Port = 8080;


// ... Create Client
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient->URL = "ws://127.0.0.1:8080";


// ... Create Protocol
TsgcWSPClient_Files oClient_Files = new TsgcWSPClient_Files();
oClient_Files->Client = oClient;


// ... Start Server
oServer->Active = true;
oClient->Connect();


// ... Send File to the client connected
oServer_Files->SendFile(FGuid, "c:\Documents\yourfile.txt");
```

Protocol Files | How Send Big Files

When you want to send big files to Server or Client, for example a File of some Gigabytes, you can experience some memory problems trying to load the full file. The Protocol Files allows you to send the files in smaller packets that when received by other peer are reassembled in a single file. Just use the **Size** parameter of **SendFile** method to set the Size in Bytes of every single packet.

```
// ... Create Server
TsgcWebSocketServer oServer = new TsgcWebSocketServer();
TsgcWSPServer_Files oServer_Files = new TsgcWSPServer_Files();
oServer_Files->Server = oServer;
oServer->Host = "127.0.0.1";
oServer->Port = 8080;


// ... Create Client
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient->URL = "ws://127.0.0.1:8080";


// ... Create Protocol
TsgcWSPClient_Files oClient_Files = new TsgcWSPClient_Files();
oClient_Files->Client = oClient;


// ... Start Server
oServer->Active = true;


// ... Connect client and Send File in packets of 100000 bytes
if oClient->Connect() then
  oClient_Files->SendFile("c:\Documents\yourfile.txt", 100000, qosLevel0, "");
```

Protocol Presence

Presence protocol allows you to know who is subscribed to a channel, this makes it easier to create chat applications and know who is online, example: game users, chat rooms, users viewing the same document...

Features

- By default user is **identified by a name**, but this can be **customized** passing more data: email, company, twitter...
- Events to **Authorize** if a **Channel** can be created, if a **member** is allowed...
- Every time a new **member joins** a channel, all members are **notified**.
- **Publish messages** to all channel subscribers.
- **Low memory** usage.

Components

TsgcWSPServer_Presence: Server Protocol Presence VCL Component.

TsgcWSPClient_Presence: Client Protocol Presence VCL Component.

Classes

TsgcWSPresenceMessage: the object which encapsulates presence packet information.

TsgcWSPServer_Presence

This is the Server Presence Protocol Component.

Introduction

This is the Server Presence Protocol Component. You need to drop this component in the form and select a TsgcWebSocketServer Component using Server Property.

Reference

- Properties — 6 properties available on the component.
- Methods — 1 public methods.
- Events — 17 events you can handle.

TsgcWSPresenceMessage

This object encapsulates all internal messages exchanged by the server and client presence protocol.

TsgcWSPresenceMember

ID: internal identifier
Name: member name, provided by the client.
Info: member additional info, provided by the client.
Data: TObject which can be used for server purposes.

TsgcWSPresenceMemberList

Member[i]: member of a list by index
Count: number of members of the list

TsgcWSPresenceChannel

Name: channel name, provided by the client.

TsgcWSPresenceMsg

Text: text message, provided by the client when call Publish method

TsgcWSPresenceError

Code: integer value identifying the error
Text: error description.

TsgcWSPClient_Presence

This is the Client Presence Protocol Component.

Introduction

This is the Client Presence Protocol Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Reference

- Properties — 7 properties available on the component.
- Methods — 5 public methods.
- Events — 16 events you can handle.

Protocol Presence Javascript

Presence Protocol Javascript sgcWebSockets uses **sgcWebSocket.js and presence.esegece.com.js** files.

Here you can find available methods, you must replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/presence.esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
</script>
```

New Member after connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcsession', function(event)
  {
    socket.newmember(event.id, 'John', 'Additional Info');
  });
</script>
```

Subscribe to Topic 1 channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.subscribe('Topic 1');
</script>
```

Unsubscribe from Topic 1 channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.unsubscribe('Topic 1');
</script>
```

Publish Message to Topic 1 channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.publish('Hello sgcWebSockets!', 'Topic 1');
</script>
```

Receive Message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcpublishmsg', function(event)
  {
    console.log('#publishmsg: ' + event.message.text);
  });
</script>
```

Get All Members Connected

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcgetmembers', function(event)
  {
    for (var i  in event.members) {
     console.log(event.members[i].id + ' ' + event.members[i].name);
    }
  });
  socket.getmembers();
</script>
```

Show Alert when Members subscribe/unsubscribe

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcnewmember', function(event)
  {
    alert('#new member: ' + event.member.name);
  }
  );
  socket.on('sgcremovemember', function(event)
  {
    alert('#removed member: ' + event.member.name);
  }
  );
  socket.on('sgcnewchannelmember', function(event)
  {
    alert('#new member: ' + event.member.name + ' in channel: ' + event.channel.name);
  }
  );
  socket.on('sgcremovechannelmember', function(event)
  {
    alert('#remove member: ' + event.member.name + ' from channel: ' + event.channel.name);
  }
  );
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('sgcerrormemberchannel', function(event)
  {
    alert('#error member channel: ' + event.error.text);
  }
  );
  socket.on('sgcerrorpublishmsg', function(event)
  {
    alert('#error publish: ' + event.error.text);
  }
  );
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  socket.close();
</script>
```

Protocol E2EE

End-to-End Encryption (E2EE) means that messages are encrypted on the sender device and can be decrypted only on recipient devices. The server routes packets but cannot read plaintext content.

This topic explains the technical flow for:

- **Direct messages (1:1)** between two peers.
- **Group messages** using a sender-key model with membership-based key rotation.

Cryptographic building blocks

- **Identity / key agreement:** Elliptic Curve Diffie-Hellman (ECDH) P-256.
- **Symmetric encryption:** AES-256-GCM.
- **Key derivation:** HKDF-SHA-256 to derive encryption keys from shared secrets.
- **Message authentication:** AEAD authentication tag (provided by AES-GCM).
- **Replay protection:** per-message nonce/IV plus sender sequence counters.

Direct messages (1:1) technical flow

1. **Public key discovery**
   Each peer publishes a public identity key. The server may distribute public keys, but private keys never leave the client device.
2. **Shared secret establishment**
   The sender and recipient perform ECDH (private key + peer public key) and obtain the same shared secret.
3. **Session key derivation**
   HKDF-SHA-256 derives one or more symmetric keys (encryption key, optional header key) from the ECDH output.
4. **Message encryption**
   The plaintext is encrypted with AES-256-GCM using a unique nonce/IV. Output includes ciphertext + authentication tag.
5. **Transport**
   The server forwards encrypted payloads and metadata (for example: sender id, key id, counter, timestamp) without plaintext access.
6. **Recipient decryption**
   The recipient derives the same session key, verifies the authentication tag, and decrypts. Any tampering causes authentication failure.

Group messages technical flow (Sender Keys)

For groups, encrypting each message separately for every member is expensive. A common optimization is a **sender key** design:

- Each sender maintains a **Sender Key State** per group.
- The state contains a symmetric **chain key** and message counter.
- For every outgoing group message, the sender derives a one-time message key from the chain key and advances the chain (hash ratchet).
- The message key encrypts the payload with AES-256-GCM.

How a sender key is distributed

1. When a sender joins a group, it creates a fresh sender key state.
2. The sender key state is shared to each current member over existing 1:1 encrypted sessions (pairwise E2EE).
3. After distribution, normal group payloads use the sender-key fast path (single encryption per message).

Membership changes and sender key rotation

To preserve forward and backward secrecy, group sender keys must rotate on membership events:

- **User joins group:** rotate sender keys so the new member cannot decrypt older history unless explicitly shared.
- **User leaves or is removed:** rotate sender keys immediately so the removed member cannot decrypt new traffic.

- **Periodic rotation:** optional time-based or message-count rotation reduces impact of key compromise.

Typical rotation sequence:

1. Create a new sender key state (new key id, new chain key).
2. Distribute it only to currently authorized members through pairwise E2EE channels.
3. Start encrypting new group messages with the new key id.
4. Accept old key id only during a short transition window, then retire it.

Security properties

- **Confidentiality:** only clients with valid keys can decrypt.
- **Integrity and authenticity:** AEAD verification detects tampering and forged ciphertexts.
- **Server blindness:** relay servers do not hold plaintext keys for message content.
- **Post-membership protection:** rotated keys block former members from future group messages.

Components

TsgcWSPServer_E2EE: Server Protocol E2EE component. It forwards encrypted messages between clients without knowing message contents.

TsgcWSPClient_E2EE: Client Protocol E2EE component. It manages key exchange, encryption and decryption on peer devices.

TsgcWSPServer_E2EE

The E2EE server protocol component routes encrypted direct and group messages between clients while keeping payloads opaque.

Introduction

The E2EE server protocol component routes encrypted direct and group messages between clients while keeping payloads opaque. Drop it on a form, assign a TsgcWebSocketHTTPServer (or other server component) to the **Server** property, and configure the E2EE options. The implementation lives in the unit **sgcWebSocket_Protocol_E2EE_Server**.

Configuration

To use the component:

- Set **Server** to a configured **TsgcWebSocketHTTPServer** instance.
- Configure acknowledgment handling in **E2EE_Options.Ack**.
- Choose how new public keys are handled with **E2EE_Options.PublicKeys.ReceiveNewPublicKey**.
- Set **Server.Active := True** to begin listening.

Reference

- Properties — 5 properties available on the component.
- Methods — 0 public methods.
- Events — 8 events you can handle.

TsgcWSPClient_E2EE

The E2EE client protocol component adds end-to-end encryption over a WebSocket connection, including encrypted direct messages and encrypted group messages.

Introduction

The E2EE client protocol component adds end-to-end encryption over a WebSocket connection, including encrypted direct messages and encrypted group messages. Drop the component on a form, assign a TsgcWebSocketClient to the **Client** property, and configure the E2EE options before connecting. The implementation lives in the unit **sgcWebSocket_Protocol_E2EE_Client**.

Configuration

To use the component:

- Set **Client** to a configured **TsgcWebSocketClient** instance.
- Assign a unique user identifier using **E2EE_Options.UserId**.
- Enable message acknowledgments with **E2EE_Options.Ack.RcvDirectMessage** and **E2EE_Options.Ack.RcvGroupMessage** when you want delivery state callbacks.
- Set **Client.Active := True** to open the WebSocket connection.

Reference

- Properties — 5 properties available on the component.
- Methods — 9 public methods.
- Events — 19 events you can handle.

WebSocket APIs

There are several implementations based on WebSockets: finance, message publishing, queues... sgcWebSockets implements the most important APIs based on WebSocket protocol. In order to use an API, just attach API component to the client and all messages will be handled by API component (only one API component can be attached to a client).

Client APIs

API	Description
Binance	is an international multi-language cryptocurrency exchange.
Binance Futures	allows you to connect to Binance Futures WebSocket / REST Market Streams.
Coinbase	Coinbase is a US-based crypto exchange. Trade Bitcoin (BTC), Ethereum (ETH), a Support for WebSocket API and REST API.
SignalR	is a library for ASP.NET developers that makes developing real-time web functio transport.
SignalRCore	ASP.NET Core SignalR is an open-source library that simplifies adding real-time we
SocketIO	is a JavaScript library for real-time web applications. It enables real-time, bi-dire web clients and servers.
Kraken	is a US-based cryptocurrency exchange.
Kraken Futures	allows you to connect to Kraken Futures WebSocket / REST Market data.
Pusher	Pusher is an easy and reliable platform with flexible pub/sub messaging, live us
Bitfinex	Bitfinex is one of the world's largest and most advanced cryptocurrency trading Bitcoin, Ethereum, Ripple, EOS, Bitcoin Cash, Iota, NEO, Litecoin, Ethereum Class
Bitstamp	Bitstamp is one of the world's longest standing crypto exchange, supporting the 2011.
Huobi	is an international multi-language cryptocurrency exchange.
Cex	is a cryptocurrency exchange and former Bitcoin cloud mining provider.
Cex Plus	CEX.IO Exchange Plus is the ultimate crypto trading platform that features deep tools
Bitmex	is a cryptocurrency exchange and derivative trading platform.
3Commas	It's a crypto trading bot.
Kucoin	is a cryptocurrency exchange that allows you to buy, sell, and store cryptocurre DOGE
Kucoin Futures	allows you to connect to Kucoin Futures Servers (WebSocket and REST)
OKX	formerly known as OKEx, is one of the largest crypto spot and derivatives tradir
XTB	FX and CFD trading, providing access to over +2000 financial markets.
Discord	is one of the most popular communication tools for online gaming and streami
Bybit	cryptocurrency exchange and trading platform
OpenAI API	The OpenAI Realtime API enables low-latency, multimodal interactions including al experiences and real-time transcription.
MEXC	Centralized cryptocurrency exchange and trading platform, this component imp
MEXC Futures	Centralized cryptocurrency exchange and trading platform, this component imp
Bitget	Cryptocurrency exchange and trading platform supporting Spot and Futures ma
Gate.io	Cryptocurrency exchange supporting Spot and Futures trading with WebSocket
Deribit	Cryptocurrency derivatives exchange offering futures and options trading on Bi
Crypto.com	Cryptocurrency exchange supporting Market and User channels with WebSocke

HTX	International cryptocurrency exchange (formerly Huobi) with REST API for mark

WebSocket APIs can be registered at **runtime**, just call Method **RegisterAPI** and pass API component as a parameter.

Other Client APIs

API	Description
Telegram	is a cloud-based instant messaging and voice over IP service. Users can send mes videos, stickers, audio and files of any type.
Whatsapp	is an internationally available American freeware, cross-platform centralized insta service
RCON	is a TCP/IP-based communication protocol which allows console commands to be
CryptoHopper	It's a crypto trading bot and portfolio manager.
CryptoRobotics	It's a crypto trading robot.

Server APIs

API	Description
WebPush	The Web Push protocol allows web applications to send notifications to not open or active.
WebAuthn	FIDO2/WebAuthn server API for passwordless authentication using bion

Errors in HTTP Requests

The API components send their REST requests through a common HTTP client. When a request fails with an HTTP error status, the client raises **EsgcHTTPAPIProtocolException**, declared in the unit sgcHTTP_API. This exception descends from the exception raised in previous versions, so existing exception handlers keep working without any change, and it carries the response headers of the failed request. Before, the HTTP object was destroyed before the error reached the application and those headers were lost, so a Retry-After header on an HTTP 429, or the rate-limit counters returned by the exchanges, could not be read at all.

- **ErrorCode:** the HTTP status code returned by the server.
- **ResponseHeaders:** the complete list of response headers.
- **GetHeader:** the value of a single response header, an empty string when the header is not present.
- **RetryAfterMs:** the Retry-After delay in milliseconds. It understands the seconds form and the HTTP-date form, and it returns -1 when the header is not present.

The ResponseHeaders parameter accepted by the Post and Query methods is now filled when the request fails too, not only when it succeeds.

Read Binance | Get Market Data for a complete example.

Throttle

Every WebSocket API client has a **Throttle** property, which limits the messages the client sends to the server. Exchanges close a connection that exceeds their message rate, so a burst of subscribe frames can drop the socket.

- **Enabled:** paces the messages sent by the application. By default false, so the timing of the calls the application makes does not change unless it is switched on.
- **MaxMessages:** maximum number of messages sent inside every interval. By default 4. The OKX client uses 3, which is the limit that exchange documents.
- **IntervalMs:** length of the window in milliseconds. By default 1000.
- **PaceBatch:** paces the frames of a multi-stream batch subscription. A batch is split into as many frames as the venue's per-frame limits require, so that burst is generated inside the component and not by the application, and it is paced by default. Set it to false to write the whole burst back to back.
- **PaceResubscribe:** paces the replay of the accumulated subscriptions after a reconnection. By default true, because that replay is generated inside the component and the application cannot avoid it. Set it to false to restore the previous behaviour.

Resubscribe

The WebSocket API clients which offer a **Resubscribe** option replay the subscriptions that were active before a disconnection as soon as the **WatchDog** reconnects. When the client holds credentials, the venue authenticates the new session asynchronously, so the replay is held until the venue acknowledges the authentication, and it is cancelled when that acknowledgement reports a failure. Before this, the private channels were replayed on a session the venue did not consider authenticated yet and the subscriptions were rejected.

The clients which wait for the acknowledgement are **Bitfinex**, **Bitget**, **Bitmex**, **Bybit**, **Cex Plus**, **Crypto.com**, **Deribit** and **Gate.io**. **Kraken Futures** does the same through its signing challenge, the private feeds are replayed once the new challenge arrives and they are re-signed with it.

A public session, where no ApiKey is set, is not authenticated at all, so its replay is sent as soon as the connection is established, exactly as before.

API Binance

Binance

Binance is an international multi-language cryptocurrency exchange. It offers some APIs to access Binance data. The following APIs are supported:

1. **WebSocket streams:** allows you to subscribe to some methods and get data in real-time. Events are pushed to clients by server to subscribers. Uses WebSocket as protocol.
2. **UserData stream:** subscribed clients get account details. Requires an API key to authenticate and uses WebSocket as protocol.
3. **REST API:** Requires an API Key and Secret to authenticate and uses HTTPs as protocol.
    1. **Market Data**
    2. **Account and Trading Data**
    3. **Wallet**
4. **Futures:** WebSocket Futures Market Data Streams are supported through the Binance Futures Client API.

The client supports **Binance**.us too, the following APIs are supported:

1. **WebSocket streams:** allows you to subscribe to some methods and get data in real-time. Events are pushed to clients by server to subscribers. Uses WebSocket as protocol.
2. **UserData stream:** subscribed clients get account details. Requires an API key to authenticate and uses WebSocket as protocol.
3. **REST API:** clients can request to server market and account data. Requires an API Key and Secret to authenticate and uses HTTPs as protocol.

Properties

Binance API has 2 types of methods: public and private. Public methods can be accessed without authentication, for example: get ticker prices. Some are private and related to user data; those methods require the use of Binance API keys.

- **ApiKey:** you can request a new api key in your binance account, just copy the value to this property.
- **ApiSecret:** API secret is only required for REST_API, websocket api only requires ApiKey for some methods.
- **TestNet:** if enabled it will connect to Binance Demo Account (by default false).
    - **HTTPLogOptions:** stores in a text file a log of HTTP requests
        - **Enabled:** if enabled, will store all HTTP requests of WebSocket API.
        - **FileName:** full path of filename where logs will be stored
        - **REST:** stores in a text file a log of REST API requests
            - **Enabled:** if enabled, will store all HTTP Requests of REST API.
            - **FileName:** full path of filename where logs will be stored.
- **UserStream:** if enabled the client will receive notifications on Account, Orders or Balance Updates (by default true).
    - On Binance.com Spot and on the Spot TestNet, the User Data Stream is obtained from the Binance WebSocket API. The component opens its own connection to wss://ws-api.binance.com:443/ws-api/v3 (wss://ws-api.testnet.binance.vision/ws-api/v3 if TestNet is enabled) and subscribes there, while your client stays connected to the market data endpoint as before. The events are received exactly as before, so there is nothing to change in the code that handles them. The subscription is signed, so Binance.ApiSecret must now be set as well as Binance.ApiKey, where the previous ListenKey request needed only the ApiKey. An application that left ApiSecret empty must now fill it in.
    - On Binance.us the User Data Stream still uses a ListenKey.
- **BinanceUS:** if enabled, will connect to Binance.us Servers (instead of Binance.com servers which is the default).
- **ListenKeyOnDisconnect:** this property specifies what to do when the client disconnect from Binance servers with an Active ListenKey. It only applies to Binance.us and to Binance Futures, because Binance.com Spot no longer uses a ListenKey.
    - **blkodDeleteListenKey:** Delete the Active ListenKey doing an HTTP Request to Binance Servers (this is the default).

    - **blkodClearListenKey:** Doesn't delete the ListenKey from Binance Servers and just clears the value of the field.
    - **blkodDoNothing:** does nothing, so the next time that connects to Binance will try to use the same ListenKey.
- **UseCombinedStreams:** if enabled, will combine streams as follows: {"stream":"<streamName>","data":<rawPayload>} (by default disabled)
- **UseServerTimeOffset:** if enabled, the timestamp of the signed WebSocket API requests is corrected with the drift measured by the REST client, so a client whose clock is not synchronized does not get its signed WebSocket requests rejected (by default false). Call **REST_API.SyncServerTime** first, the correction is zero until the drift has been measured. Read Private Requests Time.
- **RecvWindow:** the recvWindow, in milliseconds, sent and signed with the WebSocket API user data stream subscribe request. Zero leaves it out and Binance applies its own default (by default 0).
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).
- **Throttle:** limits the messages the client sends to Binance, which closes a connection that sends more than 5 messages per second.
    - **Enabled:** paces the messages sent by the application (by default false).
    - **MaxMessages:** maximum number of messages sent inside every interval (by default 4).
    - **IntervalMs:** length of the window in milliseconds (by default 1000).
    - **PaceBatch:** paces the frames a multi-stream batch subscription is split into, for example a **SubscribeStreams** call with more streams than fit in a single frame (by default true).
    - **PaceResubscribe:** paces the replay of the accumulated subscriptions after a reconnection (by default true).

Most common uses

- **WebSockets API**
    - **How to Connect to WebSocket API**
    - **How to Subscribe to a WebSocket Channel**
- **REST API**
    - **How to Get Market Data**
    - **How to Use Private REST API**
    - **How to Trade Spot**
    - **Private Requests Time**
    - **Withdraw**

WebSocket Stream API

Base endpoint is wss://stream.binance.com:9443, client can subscribe / unsubscribe from events after a successful connection.
The following Subscription / Unsubscription methods are supported.

Method	Parameters	Description
AggregateTrades	Symbol	push trade information that is aggregated for a single taker order
Trades	Symbol	push raw trade information; each trade has a unique buyer and seller
KLine	Symbol, Interval	push updates to the current klines/candlestick every second, minute, hour...
MiniTicker	Symbol	24hr rolling window mini-ticker statistics. These are NOT the statistics of the UTC day, but a 24hr rolling window for the previous 24hrs.
AllMiniTickers		24hr rolling window mini-ticker statistics for all symbols that changed in an array. These are NOT the statistics of the UTC day, but a 24hr rolling window for the previous 24hrs. Note that only tickers that have changed will be present in the array.

Ticker	Symbol	24hr rolling window ticker statistics for a single symbol. These are NOT the statistics of the UTC day, but a 24hr rolling window for the previous 24hrs.
AllMarketTickers		24hr rolling window ticker statistics for all symbols that changed in an array. These are NOT the statistics of the UTC day, but a 24hr rolling window for the previous 24hrs. Note that only tickers that have changed will be present in the array.
BookTicker	Symbol	Pushes any update to the best bid or ask's price or quantity in real-time for a specified symbol.
AllBookTickers		Pushes any update to the best bid or ask's price or quantity in real-time for all symbols.
PartialBookDepth	Symbol, Depth	Top <levels> bids and asks, pushed every second. Valid <levels> are 5, 10, or 20.
DiffDepth	Symbol	Order book price and quantity depth updates used to locally manage an order book.

Several streams can also be subscribed by name with a single frame. Call **SubscribeStreams** or **UnsubscribeStreams** with an array of stream names, read How to Subscribe to a WebSocket Channel.

After a successful subscription / unsubscription, client receives a message about it, where id is the result of Subscribed / Unsubscribed method.

```
{
  "result": null,
  "id": 1
}
```

User Data Stream API

Requires a valid ApiKey obtained from your binance account, and ApiKey must be set in Binance.ApiKey property of component.

The following data is pushed to client every time there is a change. There is no need to subscribe to any method, this is done automatically if you set a valid ApiKey.

Binance retired the Spot ListenKey REST endpoints (POST, PUT and DELETE /api/v3/userDataStream) and the wss://stream.binance.com:9443/ws/<listenKey> connection on 2026-02-20. On Binance.com Spot and on the Spot TestNet, the client subscribes to the User Data Stream on the Binance WebSocket API instead, using a second connection that the component opens to wss://ws-api.binance.com:443/ws-api/v3 (wss://ws-api.testnet.binance.vision/ws-api/v3 if TestNet is enabled). Enable Binance.UserStream as before, the subscription is renewed automatically after a reconnection and the events are received in the same shape, so existing handlers keep working. The new subscription is signed, so Binance.ApiSecret must now be set in addition to Binance.ApiKey, where the retired ListenKey request needed only the ApiKey. On Binance.us the ListenKey mechanism is still used.

Method	Description
Account Update	Account state is updated with the outboundAccountInfo event.
Balance Update	Balance Update occurs during the following: • Deposits or withdrawals from the account • Transfer of funds between accounts (e.g. Spot to Margin)
Order Update	Orders are updated with the executionReport event.

REST API

The base endpoint is: https://api.binance.com. All endpoints return either a JSON object or array. Data is returned in ascending order. Oldest first, newest last.

Access to the REST API Options, using the property REST_API.BinanceOptions.

Public API EndPoints

These endpoints can be accessed without any authorization.

General EndPoints

Method	Parameters	Description
Ping		Test connectivity to the Rest API.
GetServerTime		Test connectivity to the Rest API and get the current server time.
GetExchange-Information		Current exchange trading rules and symbol information

Market Data EndPoints

Method	Parameters	Description
GetOrderBook	Symbol	Get Order Book.
GetTrades	Symbol	Get recent trades
GetHistorical-Trades	Symbol	Get older trades.
GetAggregate-Trades	Symbol	Get compressed, aggregate trades. Trades that fill at the time, from the same order, with the same price will have the quantity aggregated.
GetKLines	Symbol, Interval	Kline/candlestick bars for a symbol. Klines are uniquely identified by their open time.
GetAver-agePrice	Symbol	Current average price for a symbol.
Get24hrTicker	Symbol	24 hour rolling window price change statistics. Careful when accessing this with no symbol.
GetPriceTicker	Symbol	Latest price for a symbol.
GetPriceTickers	Symbols	Latest price for an array of symbols. Example: ["BTCUSDT","BNBUSDT"]
GetBookTicker	Symbol	Best price/qty on the order book for a symbol or symbols.
GetUIKLines	Symbol, Interval	Kline/candlestick bars for a symbol. The response is similar to GetKLines, optimized for presentation of candlestick charts.
GetRollingWin-dowTicker	Symbol, Symbols, WindowSize	Rolling window price change statistics. Note: WindowSize default is 1d if not specified.
GetTrading-DayTicker	Symbol, Symbols, Type	Price change statistics for a trading day.

Private API EndPoints

Requires an APIKey and APISecret to get authorized by server.

Account Data EndPoints

Method	Parameters	Description
NewOrder	Symbol, Side, Type	Send in a new order.
PlaceMarketOrder	Side, Symbol, Quantity	Places a New Market Order
PlaceMarketQuoteOrder	Side, Symbol, QuoteOrderQty	Places a New Market Quote Order
PlaceLimitOrder	Side, Symbol, Quantity, LimitPrice	Places a New Limit Order
PlaceStopOrder	Side, Symbol, Quantity, StopPrice, LimitPrice	Places a New Stop Order
PlaceStopTrailingOrder	Side, Symbol, Quantity, TrailingDelta, LimitPrice	Places a New Stop Trailing Order
PlaceTakeProfitOrder	Side, Symbol, Quantity, StopPrice, LimitPrice	Places a New Take Profit Order
PlaceTakeProfitTrailingOrder	Side, Symbol, Quantity, TrailingDelta, LimitPrice	Places a New Take ProfitTrailing Order
PlaceLimitMakerOrder	Side, Symbol, Quantity	Places a New Limit Market Order
TestNewOrder	Symbol, Side, Type	Test new order creation and signature/recvWindow long. Creates and validates a new order but does not send it into the matching engine.
QueryOrder	Symbol	Check an order's status.
CancelOrder	Symbol	Cancel an active order. Cancel an active order. Either OrderId or OrigClientOrderId must be sent.
CancelAllOpenOrders	Symbol (optional)	
GetOpenOrders		Get all open orders on a symbol. Careful when accessing this with no symbol.
GetAllOrders	Symbol	Get all account orders; active, canceled, or filled.
NewOCO	Symbol, Side, Quantity, Price, StopPrice	Send in a new OCO
CancelOCO	Symbol	Cancel an entire Order List
QueryOCO	Symbol	Retrieves a specific OCO based on provided optional parameters
GetAllOCO		Retrieves all OCO based on provided optional parameters
GetOpenOCO		Get All Open OCO.
GetAccountInformation		Get current account information.
GetAccountTradeList	Symbol	Get trades for a specific account and symbol.
CancelReplaceOrder	Symbol, Side, Type, CancelReplaceMode	Cancels an existing order and places a new order on the same symbol.

NewOrderListOCO	Symbol, Side, Quantity, AboveType, BelowType	Place a new OCO order list.
NewOrderListOTO	Symbol, WorkingType, WorkingSide, WorkingQuantity, WorkingPrice, PendingType, PendingSide, PendingQuantity	Place a new OTO (One-Triggers-the-Other) order list.
NewOrderListOTOCO	Symbol, WorkingType, WorkingSide, WorkingQuantity, WorkingPrice, PendingSide, PendingAboveType, PendingBelowType, PendingQuantity	Place a new OTOCO (One-Triggers-a-One-Cancels-the-Other) order list.
NewSOROrder	Symbol, Side, Type, Quantity	Places an order using Smart Order Routing (SOR).
TestSOROrder	Symbol, Side, Type, Quantity	Test new order using Smart Order Routing (SOR). Creates and validates a new order but does not send it into the matching engine.
GetOrderRateLimitUsage		Displays the user's current order count usage for all intervals.
GetPreventedMatches	Symbol	Displays the list of orders that were expired because of STP (Self Trade Prevention).
GetAllocations	Symbol	Retrieves allocations resulting from SOR order placement.
GetAccountCommission	Symbol	Get current account commission rates.

Convert EndPoints

Method	Parameters	Description
GetAllConvertPairs	FromAsset, ToAsset	Query for all convertible token pairs and the tokens' respective upper/lower limits
GetConvertAssetInfo		Query for supported asset's precision information
SendConvertQuoteRequest	FromAsset, ToAsset	Request a quote for the requested token pairs
AcceptConvertQuote	QuoteId	Accept the offered quote by quote ID.
GetConvertOrderStatus	OrderId or QuoteId	Query order status by order ID.
PlaceConvertLimitOrder	BaseAsset, QuoteAsset, Side, LimitPrice	Enable users to place a limit order. baseAsset or quoteAsset can be determined via exchangeInfo endpoint. Limit price is defined from baseAsset to quoteAsset. Either baseAmount or quoteAmount is used.
CancelConvertLimitOrder	OrderId	Enable users to cancel a limit order
GetConvertLimitOpenOrders		Enable users to query for all existing limit orders
GetConvertTradeHistory	StartTime, EndTime	The max interval between startTime and endTime is 30 days.

Wallet EndPoints

(*wallet endpoints only work with production server, not demo)

Method	Description
GetWalletSystemStatus	Fetch system status.
GetWalletAllCoinsInformation	Get information of coins (available for deposit and withdraw) for user.
GetWalletDailyAccountSnapshot	Type: "SPOT", "MARGIN", "FUTURES" • The query time period must be less then 30 days • Support query within the last one month only • If startTimeand endTime not sent, return records of the last 7 days by default
SetWalletDisableFastWithdrawSwitch	This request will disable fastwithdraw switch under your account. You need to enable "trade" option for the api key which requests this endpoint.
SetWalletEnableFastWithdrawSwitch	This request will enable fastwithdraw switch under your account. You need to enable "trade" option for the api key which requests this endpoint. When Fast Withdraw Switch is on, transferring funds to a Binance account will be done instantly. There is no on-chain transaction, no transaction ID and no withdrawal fee.
WalletWithdraw	Submit a withdraw request.
GetWalletDepositHistory	Fetch deposit history.
GetWalletWithdrawHistory	Fetch Withdraw history.
GetWalletDepositAddress	Fetch deposit address with network.
GetWalletAccountStatus	Fetch account status detail.
GetWalletAccountAPITradingStatus	Fetch account api trading status detail.
GetWalletDustLog	Only return last 100 records Only return records after 2020/12/01
GetWalletAssetsConvertedBNB	
WalletDustTransfer	Convert dust assets to BNB. You need to openEnable Spot & Margin Trading permission for the API Key which requests this endpoint.
GetWalletAssetDividendRecord	Query asset dividend record.
GetWalletAssetDetail	Fetch details of assets supported on Binance.
GetWalletTradeFee	Fetch trade fee
WalletUserUniversalTransfer	You need to enable Permits Universal Transfer option for the API Key which requests this endpoint.MAIN_UMFUTURE Spot account transfer to USDⓈ-M Futures account ENUM of Type: • MAIN_CMFUTURE Spot account transfer to COIN-M Futures account • MAIN_MARGIN Spot account transfer to Margin（cross）account • UMFUTURE_MAIN USDⓈ-M Futures account transfer to Spot account • UMFUTURE_MARGIN USDⓈ-M Futures account transfer to Margin（cross）-account • CMFUTURE_MAIN COIN-M Futures account transfer to Spot account • CMFUTURE_MARGIN COIN-M Futures account transfer to Margin(cross) account • MARGIN_MAIN Margin（cross）account transfer to Spot account • MARGIN_UMFUTURE Margin（cross）account transfer to USDⓈ-M Futures • MARGIN_CMFUTURE Margin（cross）account transfer to COIN-M Futures

	• ISOLATEDMARGIN_MARGIN Isolated margin account transfer to Margin(cross) account • MARGIN_ISOLATEDMARGIN Margin(cross) account transfer to Isolated margin account • ISOLATEDMARGIN_ISOLATEDMARGIN Isolated margin account transfer to Isolated margin account • MAIN_FUNDING Spot account transfer to Funding account • FUNDING_MAIN Funding account transfer to Spot account • FUNDING_UMFUTURE Funding account transfer to UMFUTURE account • UMFUTURE_FUNDING UMFUTURE account transfer to Funding account • MARGIN_FUNDING MARGIN account transfer to Funding account • FUNDING_MARGIN Funding account transfer to Margin account • FUNDING_CMFUTURE Funding account transfer to CMFUTURE account • CMFUTURE_FUNDING CMFUTURE account transfer to Funding account
GetWallet-QueryUserUniver-salTransferHistory	• fromSymbol must be sent when type are ISOLATEDMARGIN_MARGIN and ISOLATEDMARGIN_ISOLATEDMARGIN • toSymbol must be sent when type are MARGIN_ISOLATEDMARGIN and ISOLATEDMARGIN_ISOLATEDMARGIN • Support query within the last 6 months only • If startTimeand endTime not sent, return records of the last 7 days by default
GetWalletFundingWal-let	Currently supports querying the following business assets：Binance Pay, Binance Card, Binance Gift Card, Stock Token
GetWalletUserAsset	Get user assets, just for positive data.
GetWalletApiKeyPer-mission	

Events

Binance Messages are received in TsgcWebSocketClient component, you can use the following events:

OnConnect
After a successful connection to Binance server.
OnDisconnect
After a disconnection from Binance server
OnMessage
Messages sent by server to client are handled in this event.
OnError
If there is any error in protocol, this event will be called.
OnException
If there is an unhandled exception, this event will be called.

Additionally, there is a specific event in Binance API Component, called **OnBinanceHTTPException**, which is raised every time there is an error calling an HTTP Request (REST API or WebSocket User Stream).

The exception raised when an HTTP request fails is **EsgcHTTPAPIProtocolException**, which carries the status code and the response headers returned by Binance, so values like Retry-After or X-MBX-USED-WEIGHT-1M can be read. Check How to Get Market Data for an example.

When the private User Data Stream cannot be opened, the error reported includes the HTTP status code and the message returned by Binance, so the reason is visible without assigning OnBinanceHTTPException.

(*) Due to changes in Binance Servers, Indy versions before Rad Studio 10.1, won't be able to connect to Test Servers. This issue doesn't affect to Enterprise Edition or if the Indy version has been upgraded to latest.

Binance | Connect WebSocket API

In order to connect to Binance WebSocket API, just create a new Binance API client and attach to TsgcWebSocket-Client.
See below an example:

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(NULL);
oBinance->Client = oClient;
oClient->Active = true;
```

Binance | Subscribe WebSocket Channel

Binance offers a variety of channels where you can subscribe to get real-time updates of market data, orders... Find below a sample of how to subscribe to a Ticker:

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(NULL);
oBinance->Client = oClient;
oBinance->SubscribeTicker("bnbbtc");

void OnMessage(TsgcWSConnection *Connection, const string aText)
{
// here you will receive the ticker updates
}
```

Subscribe to Several Streams with a Single Frame

Binance closes a connection that sends more than 5 messages per second, so subscribing to a watchlist of several symbols by calling the individual Subscribe methods in a loop can be enough to lose the connection. Call **SubscribeStreams** with an array of stream names and the client sends a single frame instead of one frame per stream. **UnsubscribeStreams** works the same way. Both methods return the identifier of the message sent to the server.

```
oBinance->SubscribeStreams(ARRAYOFCONST(("btcusdt@aggTrade", "btcusdt@depth", "ethusdt@aggTrade")));

// and to unsubscribe
oBinance->UnsubscribeStreams(ARRAYOFCONST(("btcusdt@aggTrade", "btcusdt@depth")));
```

If the application still needs to send many messages in a short time, enable the **Throttle** option of the API client, which limits how many messages the client sends inside a time window. Read WebSocket APIs for the details.

Binance | Get Market Data

Binance offers public Market Data through REST Endpoints, when you call one of these endpoints, you will get a snapshot of the market data requested.
The Market Data Endpoints don't require authentication, so are freely available to all users.

Example: to get a snapshot of the ticker BNBBTC, make the following call:

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
ShowMessage(oBinance->REST_API->GetPriceTicker("BNBBTC"));
```

Errors and Response Headers

When a request fails with an HTTP error status, the client raises **EsgcHTTPAPIProtocolException**, declared in the unit sgcHTTP_API. The exception descends from the one raised in previous versions, so existing exception handlers keep working without any change, and it carries the response headers of the failed request. Those headers were lost before, because the HTTP object was destroyed before the error reached the application.

- **ErrorCode:** the HTTP status code returned by the server.
- **ResponseHeaders:** the complete list of response headers.
- **GetHeader:** the value of a single response header, an empty string when the header is not present.
- **RetryAfterMs:** the Retry-After delay in milliseconds. It understands the seconds form and the HTTP-date form, and it returns -1 when the header is not present.

Example: when Binance answers with an HTTP 429, wait the time the server asks for and read the request weight already consumed.

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
try
{
  ShowMessage(oBinance->REST_API->GetAggregateTrades("BTCUSDT"));
}
catch (EsgcHTTPAPIProtocolException &E)
{
  if (E.ErrorCode == 429)
    Sleep(E.RetryAfterMs());
  ShowMessage(E.GetHeader("X-MBX-USED-WEIGHT-1M"));
}
```

The ResponseHeaders parameter accepted by the Post and Query methods is now filled when the request fails too, not only when it succeeds.

Binance | Private REST API

The Binance REST API offers public and private endpoints. The Private endpoints require that messages are signed to increase the security of transactions.
First you must login to your Binance account and create a new API, you will get the following values:

- ApiKey
- ApiSecret

These fields must be configured in the Binance property of the Binance API client component.
Once configured, you can start to do private requests to the Binance Pro REST API

*Private Requests, require that your local machine has the local time synchronized, if not, the requests will be rejected by Binance server. Check the following article about this, Binance Private Requests Time.

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
oBinance->Binance->ApiKey = "<your api key>";
oBinance->Binance->ApiSecret = "<your api secret>";
ShowMessage(oBinance->REST_API->GetAccountInformation());
```

Binance | Trade Spot

Binance allows you to trade spot using its REST API.

Configuration

First you must create an **API Key** in your binance account and add privileges to trading with Spot.
Once this is done, you can start spot trading.
First, **set your ApiKey and your ApiSecret** in the Binance Client Component, this will be used to sign the requests sent to Binance server.

Place an Order

To place a new order, just call the method **REST_API.NewOrder** of the Binance Client Component.

Depending on the type of order (market, limit...) the API requires more or fewer fields.

Mandatory Fields

- **Symbol:** the product id symbol, example: BNBBTC
- **Side:** BUY or SELL
- **type:** the order type
  - LIMIT
  - MARKET
  - STOP_LOSS
  - STOP_LOSS_LIMIT
  - TAKE_PROFIT
  - TAKE_PROFIT_LIMIT
  - LIMIT_MAKER

Additional Mandatory Fields based on Type

- **LIMIT:** timeInForce, quantity, price
- **MARKET:** quantity or quoteOrderQty
- **STOP_LOSS / TAKE_PROFIT:** quantity, stopPrice
- **STOP_LOSS_LIMIT / TAKE_PROFIT_LIMIT:** timeInForce, quantity, price, stopPrice
- **LIMIT_MAKER:** quantity, price

When you send an order, there are 2 possibilities:

1. **Successful:** the function NewOrder returns the message sent by binance server.
2. **Error:** the exception is returned in the event OnBinanceHTTPException.

Place Market Order 1 BNBBTC

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
oBinance->Binance->ApiKey = "<api key>";
oBinance->Binance->ApiSecret = "<api secret>";
ShowMessage(oBinance->REST_API->NewOrder("BNBBTC", "BUY", "MARKET", "", 1));
```

Place Limit Order 1 BNBBTC at 0.009260

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
oBinance->Binance->ApiKey = "<api key>";
```

```
oBinance->Binance->ApiSecret = "<api secret>";
ShowMessage(oBinance->REST_API->NewOrder("BNBBTC", "BUY", "LIMIT", "GTC", 1, 0, 0.009260));
```

-

Binance | Private Requests Time

When you do a private request to Binance, the message is signed to increase the security of requests. The message takes the local time and sends inside the signed message, if the local time has a difference greater than 5 seconds with Binance servers, the request will be rejected. So, it's important to verify that your local time is synchronized, you can do this using the synchronization time method for your OS.

The logic is as follows:

```
if (timestamp < (serverTime + 1000) && (serverTime - timestamp) <= recvWindow) {
  // process request
} else {
  // reject request
}
```

It is recommended to use a small recvWindow of 5000 or less! The max cannot go beyond 60000 milliseconds.

You can check the Binance server time, calling method **GetServerTime**, which will return the time of the Binance server

The **RecvWindow** defaults to **5000**, this value can be increased using the property **REST_API.BinanceOptions.RecvWindow**.
If your application cannot guarantee the local clock stays synchronized, for example a headless server with no reliable time service, you can let the component correct for the drift instead of having every private request rejected. Set **REST_API.BinanceOptions.UseServerTimeOffset** to **True** (by default **False**) and call the new **SyncServerTime** method whenever you want to measure the difference between the Binance server clock and your local clock, for example once at startup or on a schedule of your own choosing. SyncServerTime calls GetServerTime, compares it against the local clock, and stores the difference. From then on, every signed request's timestamp includes that stored offset.
The measured difference is available through the read only property **ServerTimeOffset**, in milliseconds, positive when the Binance server clock is ahead of your local clock. Nothing is measured or applied automatically, SyncServerTime must be called explicitly and UseServerTimeOffset must be enabled, so a signed request is only ever adjusted when the application asks for it.

Binance | Withdraw

Binance allows you to use the Wallet API to submit a Withdraw request, only the following parameters are mandatory:

- Coin
- Address
- Amount

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
oBinance->Binance->ApiKey = "<your api key>";
oBinance->Binance->ApiSecret = "<your api secret>";
ShowMessage(oBinance->REST_API->WalletWithdraw("BTC", "7213fea8e94b4a5593d507237e5a555b", 0.25));
```

API Binance Futures

Binance

Binance is an international multi-language cryptocurrency exchange. It offers some APIs to access Binance data. This component allows you to get Binance Futures WebSocket Market Streams.

https://binance-docs.github.io/apidocs/futures/en
https://binance-docs.github.io/apidocs/delivery/en

Futures Contracts

Binance API has 2 types of methods: public and private. Public methods can be accessed without authentication, example: get ticker prices. Some are private and related to user data; those methods require the use of Binance API keys.

- **ApiKey:** you can request a new api key in your binance account, just copy the value to this property.
- **ApiSecret:** API secret is only required for REST_API, websocket api only requires ApiKey for some methods.
- **TestNet:** if enabled it will connect to Binance Demo Account (by default false).
    - **HTTPLogOptions:** stores in a text file a log of HTTP requests
        - **Enabled:** if enabled, will store all HTTP requests of WebSocket API.
        - **FileName:** full path of filename where logs will be stored
        - **REST:** stores in a text file a log of REST API requests
            - **Enabled:** if enabled, will store all HTTP Requests of REST API.
            - **FileName:** full path of filename where logs will be stored.
- **UserStream:** if enabled the client will receive notifications on Account, Orders or Balance Updates (by default true). On USD-M Futures connected to production servers, the User Data Stream is opened on the routed /private path, this is done automatically by the component. COIN-M Futures and the Futures TestNet keep the previous URLs.
- **ListenKeyOnDisconnect:** this property specifies what to do when the client disconnect from Binance servers with an Active ListenKey.
    - **blkodDeleteListenKey:** Delete the Active ListenKey doing an HTTP Request to Binance Servers (this is the default).
    - **blkodClearListenKey:** Doesn't delete the ListenKey from Binance Servers and just clear the value of the field.
    - **blkodDoNothing:** does nothing, so the next time that connects to Binance will try to use the same ListenKey.
- **UseCombinedStreams:** if enabled, will combine streams as follows: {"stream":"<streamName>","data":<rawPayload>} (by default disabled)
- **UseServerTimeOffset:** if enabled, the timestamp of the signed WebSocket API requests is corrected with the drift measured by the REST client, so a client whose clock is not synchronized does not get its signed WebSocket requests rejected (by default false). Call **REST_API.SyncServerTime** first, the correction is zero until the drift has been measured. Read Private Requests Time.
- **RecvWindow:** the recvWindow, in milliseconds, sent and signed with the WebSocket API user data stream subscribe request. Zero leaves it out and Binance applies its own default (by default 0).
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).
- **FuturesStreamEndpoint:** selects which USD-M market data endpoint the client connects to. Binance decommissioned the legacy USD-M WebSocket URLs on 2026-04-23 and now serves the market data streams from two different endpoints, so a stream can only be subscribed on the endpoint that delivers it.
    - **bfsePublic:** delivers the BookTicker, AllBookTickers, PartialBookDepth and DiffDepth streams (this is the default).
    - **bfseMarket:** delivers the AggregateTrades, MarkPrice, AllMarkPrice, KLine, MiniTicker, AllMiniTicker, Ticker, AllMarketTickers, LiquidationOrders and AllLiquidationOrders streams.

Client can connect to USDT or COIN Binance Futures, set which contract you want to trade using **FuturesContracts** property:

- **bfcUSDT:** connects to USD-M Futures API.
- **bfcCOIN:** connects to COIN-M Futures API.

Client can connect to Production or Demo Binance accounts. If **TestNet** property is enabled, it will connect to Demo account, otherwise will connect to production Binance Servers.

The **FuturesStreamEndpoint** property only affects USD-M Futures connected to production servers, COIN-M Futures and the Futures TestNet are not routed and are unaffected. Its default value keeps the behaviour of previous versions, so if you subscribe to KLine, MarkPrice, Ticker, AggregateTrades, MiniTicker or LiquidationOrders you must set Binance.FuturesStreamEndpoint to bfseMarket before connecting. If you subscribe to a stream that is not delivered by the selected endpoint, an error is raised naming the property, instead of a connection that silently delivers nothing.

WebSocket Stream API

Client can subscribe / unsubscribe from events after a successful connection.
The following Subscription / Unsubscription methods are supported.

Method	Parameters	Description
AggregateTrades	Symbol	The Aggregate Trade Streams push trade information that is aggregated for a single taker order every 100 milliseconds.
MarkPrice	Symbol, UpdateSpeed	Mark price and funding rate for a single symbol pushed every 3 seconds or every second.
AllMarkPrice	UpdateSpeed	Mark price and funding rate for all symbols pushed every 3 seconds or every second.
KLine	Symbol, Interval	The Kline/Candlestick Stream push updates to the current klines/candlestick every 250 milliseconds (if existing).
MiniTicker	Symbol	24hr rolling window mini-ticker statistics for a single symbol. These are NOT the statistics of the UTC day, but a 24hr rolling window from requestTime to 24hrs before.
AllMiniTicker		24hr rolling window mini-ticker statistics for all symbols. These are NOT the statistics of the UTC day, but a 24hr rolling window from requestTime to 24hrs before. Note that only tickers that have changed will be present in the array.
Ticker	Symbol	24hr rolling window ticker statistics for a single symbol. These are NOT the statistics of the UTC day, but a 24hr rolling window from requestTime to 24hrs before.
AllMarketTickers		24hr rolling window ticker statistics for all symbols. These are NOT the statistics of the UTC day, but a 24hr rolling window from requestTime to 24hrs before. Note that only tickers that have changed will be present in the array.
BookTicker	Symbol	Pushes any update to the best bid or ask's price or quantity in real-time for a specified symbol.
AllBookTickers		Pushes any update to the best bid or ask's price or quantity in real-time for all symbols.
LiquidationOrders	Symbol	The Liquidation Order Streams push force liquidation order information for specific symbol
AllLiquidationOrders		The All Liquidation Order Streams push force liquidation order information for all symbols in the market.
PartialBookDepth	Symbol, Depth	Top bids and asks, Valid are 5, 10, or 20.
DiffDepth	Symbol	Bids and asks, pushed every 250 milliseconds, 500 milliseconds, 100 milliseconds or in real time(if existing)

After a successful subcription / unsubscription, client receives a message about it, where id is the result of Subscribed / Unsubscribed method.

```
{
  "result": null,
  "id": 1
}
```

User Data Stream API

Requires a valid ApiKey obtained from your binance account, and ApiKey must be set in Binance.ApiKey property of component.

The following data is pushed to client every time there is a change. There is no need to subscribe to any method, this is done automatically if you set a valid ApiKey.

Method	Description
Margin Call	When the user's position risk ratio is too high, this stream will be pushed. This message is only used as risk guidance information and is not recommended for investment strategies. In the case of a highly volatile market, there may be the possibility that the user's position has been liquidated at the same time when this stream is pushed out.
Balance and Position Update	Balance Update occurs during the following: • When balance or position get updated, this event will be pushed. • When "FUNDING FEE" changes to the user's balance. •
Order Update	When new order created, order status changed will push such event.

REST API

All endpoints return either a JSON object or array. Data is returned in ascending order. Oldest first, newest last.

Public API EndPoints

These endpoints can be accessed without any authorization.

General EndPoints

Method	Parameters	Description
Ping		Test connectivity to the Rest API.
GetServerTime		Test connectivity to the Rest API and get the current server time.
GetExchange-Information		Current exchange trading rules and symbol information

Market Data EndPoints

Method	Parameters	Description

GetOrderBook	Symbol	Get Order Book.
GetTrades	Symbol	Get recent trades
GetHistorical-Trades	Symbol	Get older trades.
GetAggregate-Trades	Symbol	Get compressed, aggregate trades. Trades that fill at the time, from the same order, with the same price will have the quantity aggregated.
GetKLines	Symbol, Interval	Kline/candlestick bars for a symbol. Klines are uniquely identified by their open time.
Get24hrTicker	Symbol	24 hour rolling window price change statistics. Careful when accessing this with no symbol.
GetPriceTicker	Symbol	Latest price for a symbol or symbols.
GetBookTicker	Symbol	Best price/qty on the order book for a symbol or symbols.
GetMarkPrice	Symbol	Mark Price and Funding Rate
GetFundin-gRateHistory	Symbol	
GetOpenInter-est	Symbol	Get present open interest of a specific symbol.
GetOpenInter-estStatistics	Symbol, Period	
GetTopTrader-AccountRatio	Symbol, Period	
GetTopTrader-PositionRatio	Symbol, Period	
GetGlobalAc-countRatio	Symbol, Period	
GetTakerVol-ume	Symbol, Period	
GetContinu-ousKLines	Pair, ContractType, Interval	Kline/candlestick bars for a specific contract type.
GetIndex-PriceKLines	Pair, Interval	Kline/candlestick bars for the index price of a pair.
GetMarkPriceK-Lines	Symbol, Interval	Kline/candlestick bars for the mark price of a symbol.
GetPremiumIn-dexKLines	Symbol, Interval	Premium index kline bars of a symbol.
GetFundingInfo		Get funding rate info for all symbols.
GetPriceTick-erV2	Symbol	Latest price for a symbol or symbols (V2).
GetIndexInfo	Symbol	Get index info.
GetAssetIndex	Symbol	Get asset index for multi-assets mode.
GetCon-stituents	Symbol	Get index constituents.
GetDeliv-eryPrice	Pair	Get delivery price.
GetBasis	Pair, ContractType, Period	Get basis data.

Private API EndPoints

Requires an APIKey and APISecret to get authorized by server.

Account and Trades EndPoints

Method	Parameters	Description
ChangePositionMode	DualPosition	Change user's position mode (Hedge Mode or One-way Mode) on EVERY symbol
GetCurrentPositionMode		Get user's position mode (Hedge Mode or One-way Mode) on EVERY symbol
NewOrder	Symbol, Side, PositionSide, Type	Send in a new order.
PlaceMarketOrder	Side, Symbol, Quantity	
PlaceLimitOrder	Side, Symbol, Quantity, LimitPrice	
PlaceStopOrder	Side, Symbol, Quantity, StopPrice, LimitPrice	
PlaceTrailingStopOrder	Side, Symbol, Quantity, aActivationPrice, aCallbackRate	
QueryOrder	Symbol	Check an order's status.
CancelOrder	Symbol	Cancel an active order. Either OrderId or OrigClientOrderId must be sent.
CancelAllOpenOrders	Symbol	
AutoCancelAllOpenOrders	Symbol, CountDownTimer	Cancel all open orders of the specified symbol at the end of the specified countdown.
QueryCurrentOpenOrder	Symbol	
GetOpenOrders	Symbol	Get all open orders on a symbol. Careful when accessing this with no symbol.
GetAllOrders	Symbol	Get all account orders; active, canceled, or filled.
GetAccountBalance		
GetAccountInformation		Get current account information.
ChangeInitialLeverage	Symbol, Leverage	Change user's initial leverage of specific symbol market.
ChangeMarginType	Symbol, MarginType	
ModifyIsolatedPositionMargin	Symbol, Amount, Type	
GetPositionMarginChangeHistory	Symbol	
GetPositionInformation	Symbol	
GetAccountTradeList	Symbol	
GetIncomeHistory	Symbol	
GetNotionalLeverageBracket	Symbol	

TestNewOrder	Symbol, Side, PositionSide, Type	Test new order creation and signature/recvWindow long. Creates and validates a new order but does not send it into the matching engine.
ModifyOrder	Symbol	Modify an existing order.
New-BatchOrders	BatchOrders	Place multiple orders.
Modify-BatchOrders	BatchOrders	Modify multiple orders.
Cancel-BatchOrders	Symbol	Cancel multiple orders.
GetOrderA-mendment	Symbol	Get order modification history.
Countdown-CancelAll	Symbol, CountdownTime	Cancel all open orders of the specified symbol at the end of the specified countdown.
GetForce-Orders	Symbol	Get user's force liquidation orders.
GetADLQuan-tile	Symbol	Get ADL quantile estimation for positions.
GetAccountBal-anceV3		Get futures account balance (V3).
GetAccountIn-formationV3		Get current account information (V3).
GetPositionIn-formationV3	Symbol	Get current position information (V3).
GetCommis-sionRate	Symbol	Get user commission rate.
GetAccount-Config		Get current account configuration.
GetSymbolCon-fig	Symbol	Get symbol configuration.
GetOrderRate-Limit		Get user's order rate limit.
GetApiTrad-ingStatus	Symbol	Get API trading quantitative rules indicators.
ChangeMultiAs-setsMode	MultiAssetsMargin	Change user's multi-assets mode. Multi-Assets Mode: true; Single-Asset Mode: false.
GetMultiAs-setsMode		Get user's current multi-assets mode.
SetFeeBurn	FeeBurn	Change user's BNB fee burn status.
GetFeeBurn		Get user's BNB fee burn status.
CreateLis-tenKey		Start a new user data stream. The stream will close after 60 minutes unless a keepalive is sent.
KeepAliveLis-tenKey		Keepalive a user data stream to prevent a timeout.
CloseListenKey		Close a user data stream.

Events

Binance Futures Messages are received in TsgcWebSocketClient component, you can use the following events:

OnConnect
After a successful connection to Binance server.
OnDisconnect
After a disconnection from Binance server
OnMessage
Messages sent by server to client are handled in this event.
OnError
If there is any error in protocol, this event will be called.
OnException
If there is an unhandled exception, this event will be called.

Additionally, there is a specific event in Binance API Component, called **OnBinanceHTTPException**, which is raised every time there is an error calling an HTTP Request (REST API or WebSocket User Stream).

When the User Data Stream cannot be opened, the error reported includes the HTTP status code and the message returned by Binance, so the reason is visible without assigning OnBinanceHTTPException.

(*) Due to changes in Binance Servers, Indy versions before Rad Studio 10.1, won't be able to connect to Test Servers. This issue doesn't affect to Enterprise Edition or if the Indy version has been upgraded to the latest.

API Binance Futures | Trade

Binance allows you to trade futures using its REST API.

Configuration

First you must create an **API Key** in your binance account and add privileges to trading with Futures.
Once this is done, you can start to trading with futures.
First you must select if you want to trade with **USDT** or **COIN** futures, there is a property called FuturesContracts where you can set which future contract you want to trade
Then, **set your ApiKey and your ApiSecret** in the Binance Futures Client Component, this will be used to sign the requests sent to Binance server.

Place an Order

To place a new order, just call to method **REST_API.NewOrder** of Binance Futures Client Component.

Depending on the type of the order (market, limit...) the API requires more or less fields.

Mandatory Fields

- **Symbol:** the product id symbol, example: BTCUSD_210326
- **Side:** BUY or SELL
- **type:** the order type
  - LIMIT
  - MARKET
  - STOP
  - TAKE_PROFIT
  - STOP_MARKET
  - TAKE_PROFIT_MARKET
  - TRAILING_STOP_MARKET

Additional Mandatory Fields based on Type

- **LIMIT:** timeInForce, quantity, price
- **MARKET:** quantity
- **STOP/TAKE_PROFIT:** quantity, price, stopPrice
- **STOP_MARKET/TAKE_PROFIT_MARKET:** stopPrice
- **TRAILING_STOP_MARKET:** callbackRate

When you send an order, there are 2 possibilities:

1. **Successful:** the function NewOrder returns the message sent by binance server.
2. **Error:** the exception is returned in the event OnBinanceHTTPException.

API SocketIO

SocketIO

Socket.IO is a JavaScript library for real-time web applications. It enables real-time, bi-directional communication between web clients and servers. It has two parts: a client-side library that runs in the browser, and a server-side library for Node.js. Both components have a nearly identical API. Like Node.js, it is event-driven.

Messages Types

0: open (Sent from the server when a new transport is opened (recheck))

1: close (Request the close of this transport but does not shut down the connection itself.)

2: ping (Sent by the client. The server should answer with a pong packet containing the same data)
example
client sends: 2probe
server sends: 3probe

3: pong (Sent by the server to respond to ping packets.)

4: string message (actual message, client and server should call their callbacks with the data.)
example:
42/chat,[“join”,”{room:1}"]
4 is the message packet type in the engine.io protocol
2 is the EVENT type in the socket.io protocol
/chat is the data which is processed by socket.io
socket.io will fire the “join” event
will pass "room: 1" data. It is possible to omit namespace only when it is /.

5: upgrade (Before engine.io switches a transport, it tests, if server and client can communicate over this transport. If this test succeeds, the client sends an upgrade packets which requests the server to flush its cache on the old transport and switch to the new transport.)

6: noop (A noop packet. Used primarily to force a poll cycle when an incoming WebSocket connection is received.)

Properties

API: specifies SocketIO version:

ioAPI0: supports socket.io 0.* servers (selected by default)

ioAPI1: supports socket.io 1.* servers

ioAPI2: supports socket.io 2.* servers

ioAPI3: supports socket.io 3.* servers

ioAPI4: supports socket.io 4.* servers

Base64: if enabled, binary messages are received as base64.

HandShakeCustomURL: allows customizing the URL to get socket.io session.

HandShakeTimestamp: only enable if you want to send timestamp as a parameter when a new session is requested (enable this property if you try to access a gevent-socketio python server).

HandShakeAuthToken: if the server requires a token for authentication, set here the authentication token.

Namespace: allows setting a namespace when connects to the server.

Polling: disabling this property, client will connect directly to server using websocket as transport.

Parameters: allows you to set connection parameters.

EncodeParameters: if enabled, parameters are encoded.

Methods

Use the WriteData method to send messages to the socket.io server (following the Message Types section).
1. Call the method "add user" with one parameter using John as the user name.

```
WriteData("42[\"add user\", \"John\"]");
```

Events

OnHTTPRequest

Before a new websocket connection is established, the socket.io server requires the client to open a new HTTP connection to get a new session id. In some cases, the socket.io server requires authentication using HTTP headers. You can use this event to add custom HTTP headers, like Basic authorization or Bearer token authentication.

OnAfterConnect

This event is called after the socket.io connection is successful and the client can send messages to the server. Here you can subscribe to namespaces, for example.

OnHTTPConnectionSSL

When a WebSocket server requires secure connections, you can get an error message like this when a client tries to connect to the server:
Error connecting with SSL. error:XXXXXXXX:SSL routines:ssl3_read_bytes:tlsv1 alert protocol version
This error means that your client is trying to connect using a TLS version which is not supported by the server.
To resolve this error you must handle OnSSLAfterCreateHandler of WebSocket client component and set a newer TLS version.
For example: here we are setting TLS 1.2 as a protocol version.

```
void OnHTTPConnectionSSL(TObject *Sender; TIdSSLIOHandlerSocketBase *aSSLHandler)
{
  static_cast<tidssliohandlersocketopenssl*>(aSSLHandler)->SSLOptions->Method = sslvTLSv1_2;
}
```

API Coinbase

Coinbase

APIs supported

- WebSockets API: connect to a public websocket server and provides real-time market data updates.
- REST API: The REST API has endpoints for account and order management as well as public market data.

Most common uses

- **WebSockets API**
  - **How to Connect to WebSocket API**
  - **How to Subscribe to a WebSocket Channel**
- **REST API**
  - **How to Get Market Data**
  - **How to Use Private REST API**
  - **How to Place Orders**
  - **How to Use SandBox Account**
  - **Private Requests Time**

WebSockets API

The WebSocket feed is publicly available and provides real-time market data updates for orders and trades. Two endpoints are supported in production:

- **Market Data** is the feed that provides updates for both orders and trades. Most channels are now available without authentication.
- **User Order Data** provides updates for the orders of the user.

You can subscribe to the following channels:

Method	Arguments	Description
SubscribeHeart-Beat		Real-time server pings to keep all connections open
SubscribeStatus	**aProductId:** id of the product	Sends all products and currencies on a preset interval
SubscribeCandles	**aProductId:** id of the product	Real-time updates on product candles
SubscribeTicker	**aProductId:** id of the product	Real-time price updates every time a match happens
SubscribeTicker-Batch	**aProductId:** id of the product	Real-time price updates every 5000 milli-seconds
SubscribeLevel2	**aProductId:** id of the product	All updates and easiest way to keep order book snapshot
SubscribeMarket-Trades	**aProductId:** id of the product	Real-time updates every time a market trade happens
SubscribeUser	**aProductId:** id of the product	Only sends messages that include the authenticated user

SubscribeFutures-BalanceSummary		Real-time updates every time a user's futures balance changes

The User and FuturesBalanceSummary channels require authentication, so first request your API keys in your Coinbase account and then set the values in the property Coinbase of the component:

- ApiKey
- ApiSecret
- Resubscribe: if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Authentication will result in a couple of benefits:
1. Messages where you're one of the parties are expanded and have more useful fields
2. You will receive private messages, such as lifecycle information about stop orders you placed

REST API

Private Endpoints

Private endpoints are available for order management, and account management.
Before being able to sign any requests, you must create an API key via the Coinbase Pro website. The API key will be scoped to a specific profile. Upon creating a key you will have 3 pieces of information which you must remember:

- Key
- Secret
- Passphrase

The Key and Secret will be randomly generated and provided by Coinbase Pro; the Passphrase will be provided by you to further secure your API access. Coinbase Pro stores the salted hash of your passphrase for verification, but cannot recover the passphrase if you forget it.

You can restrict the functionality of API keys. Before creating the key, you must choose what permissions you would like the key to have. The permissions are:

- View - Allows a key read permissions. This includes all GET endpoints.
- Transfer - Allows a key to transfer currency on behalf of an account, including deposits and withdraws. Enable with caution - API key transfers WILL BYPASS two-factor authentication.
- Trade - Allows a key to enter orders, as well as retrieve trade data. This includes POST /orders and several GET endpoints.

Accounts

Method	Arguments	Description
ListAccounts		Get a list of trading accounts from the profile of the API key.
GetAccount	**aAccountId:** id of the account	Information for a single account. Use this endpoint when you know the account_id. API key must belong to the same profile as the account.

Orders

Method	Arguments	Description

PlaceNewOrder	**aOrder:** class that contains all possible fields of an order	Places a new order. Use only if you need to access to advanced order options.
PlaceMarketOrder	**aSide:** buy or sell **aProductId:** id of the product **aQuoteSize:** The amount of the second Asset in the Trading Pair. **aBaseSize:** The amount of the first Asset in the Trading Pair **aClient_oid:** Order ID selected by you to identify your order	Places a new Market order.
PlaceLimitOrder	**aSide:** buy or sell **aProductId:** id of the product **aQuoteSize:** The amount of the second Asset in the Trading Pair. **aBaseSize:** The amount of the first Asset in the Trading Pair **aLimitPrice:** price limit **Client_oid:** Order ID selected by you to identify your order	Places a new Limit order.
PlaceStopOrder	**aSide:** buy or sell **ProductId:** id of the product **aBaseSize:** The amount of the first Asset in the Trading Pair **StopPrice:** price of the stop **aLimitPrice:** price limit **aStopDirection:** loss or entry **Client_oid:** Order ID selected by you to identify your order	Places a new Stop Order
CancelOrder	**aOrderId:** id of the order	Cancel a previously placed order. Order must belong to the profile that the API key belongs to.
EditOrder	**aOrderId:** id of the order **aPrice:** price **aSize:** Amount	Edit an order with a specified new size, or new price
EditOrderPreview	**aOrderId:** id of the order **aPrice:** price **aSize:** Amount	Preview an edit order request with a specified new size, or new price.
ListOrders		Get a list of orders filtered by optional query parameters (product_id, order_status, etc).
GetOrder	**aOrderId:** id of the order	Get a single order by order ID.
PreviewOrder		Preview an order.
ClosePosition	**aOrderId:** id of the order **aProductId:** id of the product **aSize:** amount	Places an order to close any open positions for a specified product_id.

Market Data

Method	Arguments	Description
GetPublicProducts		Get a list of the available currency pairs for trading.
GetPublicProduct	**aProductId:** id of the product	Get information on a single product by product ID.
GetPublicProductBook	**aProductId:** id of the product	Get a list of bids/asks for a single product. The amount of detail shown can be customized with the limit parameter.

GetPublicProductCandles	**aProductId:** id of the product **aStart:** start of the time interval **aEnd:** end of the time interval **aGranularity:** The timeframe each candle represents.	Get rates for a single product by product ID, grouped in buckets.
GetTrades	**aProductId:** id of the product	Get snapshot information by product ID about the last trades (ticks) and best bid/ask.
GetTime		Get the current time from the Coinbase Advanced API.

Fills

Method	Arguments	Description
GetFillsByOrderId		Get a list of fills filtered by order id
GetFillsByProductId		Get a list of fills filtered by product id
GetFillsByTradeId		Get a list of fills filtered by trade id

Convert

Method	Arguments	Description
CreateConvertQuote		Create a convert quote between currencies.
CommitConvertTrade		Commit a convert trade.
GetConvertTrade		Get convert trade details.

Fees

Method	Arguments	Description
GetTransactionSummary		Get transaction fee summary.

Products (Authenticated)

Method	Arguments	Description
ListProducts		List available products.
GetProduct	**aProductId:** id of the product	Get a specific product.
GetProductBook	**aProductId:** id of the product	Get product order book.
GetProductCandles	**aProductId:** id of the product	Get product OHLCV candles.
GetMarketTrades	**aProductId:** id of the product	Get recent market trades.
GetBestBidAsk		Get best bid/ask prices.

Portfolios

Method	Arguments	Description
ListPortfolios		List all portfolios.
CreatePortfolio		Create a new portfolio.
DeletePortfolio		Delete a portfolio.
GetPortfolioBreakdown		Get portfolio breakdown details.
MovePortfolioFunds		Move funds between portfolios.

Perpetuals

Method	Arguments	Description
GetPerpetualsPortfolioSummary		Get perpetuals portfolio summary.
ListPerpetualsPositions		List perpetuals positions.
GetPerpetualsPosition		Get a specific perpetuals position.

Coinbase | Connect WebSocket API

In order to connect to Coinbase WebSocket API, just create a new Coinbase API client and attach to TsgcWebSocketClient. See below an example:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
oCoinbase->Client = oClient;
oClient->Active = true;
```

Coinbase | Subscribe WebSocket Channel

Coinbase offers a variety of channels where you can subscribe to get real-time updates of market data, orders... Find below a sample of how subscribe to a Ticker:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
oCoinbase->Client = oClient;
oCoinbase->SubscribeTicker("ETH-USD");

void OnCoinbaseMessage(TObject *Sender, string aType, string aRawMessage)
{
// here you will receive the ticker updates
}
```

Coinbase Pro | Get Market Data

Coinbase offers public Market Data through REST Endpoints, when you call one of these endpoints, you will get a snapshot of the market data requested.
The Market Data Endpoints don't require authentication, so are freely available to all users.

Example: to get an snapshot of the book BTC-USD, do the following call

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase(this);
ShowMessage(oCoinbase->REST_API->GetPublicProductBook("BTC-USD"));
```

Coinbase Pro | Private REST API

The Coinbase REST API offers public and private endpoints. The Private endpoints require that messages are signed to increase the security of transactions.
First you must login to your Coinbase account and create a new API, you will get the following values:

- ApiKey
- ApiSecret

These fields must be configured in the Coinbase property of the Coinbase API client component.
Once configured, you can start to do private requests to the Coinbase REST API

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase(this);
oCoinbase->Coinbase->ApiKey = "<your api key>";
oCoinbase->Coinbase->ApiSecret = "<your api secret>";
ShowMessage(oCoinbase->REST_API->ListAccounts);
```

Coinbase Pro | Private Requests Time

When you do a private request to Coinbase, the message is signed to increase the security of requests. The message takes the local time and sends inside the signed message, if the local time has a difference greater than 30 seconds with Coinbase servers, the request will be rejected. So, it's important to verify that your local time is synchronized, you can do this using the synchronization time method for your OS.

You can check the Coinbase Pro server time, calling method **GetTime**, which will return the time of the Coinbase Pro server

Coinbase Pro | Place Orders

In order to place new orders in Coinbase, you first need your API keys to access your private data. Check the following article: How to Use Private REST API.

Once you have configured your API keys, you can start to place orders

Market Order

Place a new Market Order, buy 0.002 contracts of BTC-USD

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase(this);
oCoinbase->Coinbase->ApiKey = "your api key";
oCoinbase->Coinbase->ApiSecret = "your api secret";
oCoinbase->Coinbase->ApiPassphrase = "your passphrase";
ShowMessage(oCoinbase->REST_API>PlaceMarketOrder(coisBuy, "BTC-USD", 0.002

, 0

));
```

Limit Order

Place a new Limit Order, buy 0.002 contracts of BTC-USD at price limit of 10000

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase(this);
oCoinbase->Coinbase->ApiKey = "your api key";
oCoinbase->Coinbase->ApiSecret = "your api secret";
oCoinbase->Coinbase->ApiPassphrase = "your passphrase";
ShowMessage(oCoinbase->REST_API>PlaceLimitOrder(coisBuy, "BTC-USD", 0.002,

0,

10000));
```

Coinbase Pro SandBox Account

Coinbase allows you to use a SandBox account where you can trade without real funds. This account requires creating API keys different from the production account.
To use the SandBox account, just set **Coinbase.SandBox** property to **true**, before doing any request to the REST API.

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase(this);
oCoinbase->Coinbase->ApiKey = "your api key";
oCoinbase->Coinbase->ApiSecret = "your api secret";
oCoinbase->Coinbase->SandBox = true;
ShowMessage(oCoinbase->REST_API>ListAccounts);
```

API SignalRCore

SignalRCore

ASP.NET Core SignalR is an open-source library that simplifies adding real-time web functionality to apps. Real-time web functionality enables server-side code to push content to clients instantly.

Good candidates for SignalR:

- Apps that require high-frequency updates from the server. Examples are gaming, social networks, voting, auction, maps, and GPS apps.
- Dashboards and monitoring apps. Examples include company dashboards, instant sales updates, or travel alerts.
- Collaborative apps. Whiteboard apps and team meeting software are examples of collaborative apps.
- Apps that require notifications. Social networks, email, chat, games, travel alerts, and many other apps use notifications.

SignalRCore sgcWebSockets component uses WebSocket as transport to connect to a SignalRCore server, if this transport is not supported, an error will be raised.

Hubs

SignalRCore uses hubs to communicate between clients and servers. SignalRCore provides 2 hub protocols: text protocol based on JSON and binary protocol based on MessagePack. The sgcWebSockets component only implements JSON text protocol to communicate with SignalRCore servers.
To configure which Hub client will use, just set in **SignalRCore/Hub** property the name of the Hub before the client connects to the server.

Connection

When a client opens a new connection to the server, sends a request message which contains format protocol and version. sgcWebSockets always sends format protocol as JSON. The server will reply with an error if the protocol is not supported by the server, this error can be handled using **OnSignalRCoreError** event, and if the connection is successful, **OnSignalRCoreConnect** event will be called.

When a client connects to a SignalRCore server, it can send a ConnectionId which identifies client between sessions, so if you get a disconnection client can reconnect to server passing same prior connection id. In order to get a new connection id, just connect normally to the server and you can know ConnectionId using **OnBeforeConnectEvent**. If you want to reconnect to the server and pass a prior connection id, use **ReConnect** method and pass **ConnectionId** as a parameter.

SignalRCore Protocol

The SignalR Protocol is a protocol for two-way RPC over any Message-based transport. Either party in the connection may invoke procedures on the other party, and procedures can return zero or more results or an error. Example: the client can request a method from the server and the server can request a method from the client. The following messages are exchanged between server and clients:

- **HandshakeRequest:** the client sends to the server to agree on the message format.
- **HandshakeResponse:** server replies to the client an acknowledgement of the previous HandshakeRequest message. Contains an error if the handshake failed.
- **Close:** called by client or server when a connection is closed. Contains an error if the connection was closed because of an error.
- **Invocation:** client or server sends a message to another peer to invoke a method with arguments or not.

- **StreamInvocation:** client or server sends a message to another peer to invoke a streaming method with arguments or not. The Response will be split into different items.
- **StreamItem:** is a response from a previous StreamInvocation.
- **Completion:** means a previous invocation or StreamInvocation has been completed. Can contain a result if the process has been successful or an error if there is some error.
- **CancelInvocation:** cancel a previous StreamInvocation request.
- **Ping:** is a message to check if the connection is still alive.

SignalRCore Encoding

SignalRCore allows you to use the following encodings:

- **JSON**: currently the only supported encoding.
- **MessagePack**

Currently, only JSON is supported although MessagePack can be used encoding the messages sent using an external messagepack library. See the section MessagePack below for more information.

The configuration of the Encoding Protocol is defined in the property SignalRCore.Protocol. By default the value is **srcpJSON**.

Authorization

Authentication can be enabled to associate a user with each connection and filter which users can access resources. Authentication is implemented using Bearer Tokens: the client provides an access token and the server validates this token and uses it to identify the user.
In standard Web APIs, bearer tokens are sent in an HTTP Header, but when using websockets, the token is transmitted as a query string parameter.
The following methods are supported:

srcaRequestToken

If Authentication is enabled, the flow is:

1. First tries to get a valid token from server. Opens an HTTP connection against Authentication.RequestToken.URL and do a POST using User and Password data.
2. If previous is successful, a token is returned. If not, an error is returned.
3. If token is returned, then opens a new HTTP connection to negotiate a new connection. Here, token is passed as an HTTP Header.
4. If previous is successful, opens a websocket connection and pass token as query string parameter.

- **Authentication.Enabled:** if active, authorization will be used before a websocket connection is established.
- **Authentication.Username:** the username provided to server to authenticate.
- **Authentication.Password:** the secret word provided to server to authenticate.
- **Authentication.RequestToken.PostFieldUsername:** name of field to transmit username (depends on configuration, check http javascript page to see which name is used).
- **Authentication.RequestToken.PostFieldPassword:** name of field to transmit password (depends on configuration, check http javascript page to see which name is used).
- **Authentication.RequestToken.URL:** url where token is requested.
- **Authentication.RequestToken.QueryFieldToken:** name of query string parameter using in websocket connection.

srcaSetToken

Here, you pass token directly to SignalRCore server (because token has been obtained from another server).

- **Authentication.Enabled:** if active, authorization will be used before a websocket connection is established.
- **Authentication.SetToken.Token:** token value obtained.

The Access token can be sent as a query parameter (this is the option by default) or sent as an HTTP Header as a Bearer Token. Use the property Authentication.TokenParam to configure this behaviour.

- **srctQuery:** the access_token is passed in the query url of the websocket connection.
- **srctHeader:** the access_token is passed as an http header as a Bearer Token.

srcaBasic

This option uses **Basic Authentication**, this authentication method requires configuring the SignalRCore component and the TsgcWebSocketClient.

Example: if the server requires basic authentication and the username is "user" and the password is "secret", configure the components as shown below.

```
// websocket client
TsgcWebSocketClient* WSClient = new TsgcWebSocketClient();
WSClient->Authentication->Enabled = true;
WSClient->Authentication->Basic->Enabled = true;
WSClient->Authentication->URL->Enabled = false;
WSClient->Authentication->Session->Enabled = false;
WSClient->Authentication->Token->Enabled = false;
WSClient->Authentication->User = "user";
WSClient->Authentication->Password = "secret";
// signalrcore
TsgcWSAPI_SignalRCore* Signal = new TsgcWSAPI_SignalRCore();
Signal->SignalRCore->Authentication->Enabled = true;
Signal->SignalRCore->Authentication->Authentication = srcaBasic;
Signal->SignalRCore->Authentication->Username = "user";
Signal->SignalRCore->Authentication->Password = "secret";
Signal->Client = WSClient;
```

Communication between Client and Server

There are three kinds of interactions between server and clients:

Invocations

The Caller sends a message to the Callee and expects a message indicating that the invocation has been completed and optionally a result of the invocation

Example: client invokes SendMessage method and passes as parameters user name and text message. Sends an Invocation Id to
get a result message from the server.

```
SignalRCore->Invoke("SendMessage", ARRAYOFCONST(("John", "Hello All.")), "id-000001");

void OnSignalRCoreCompletion(TObject *Sender, TSignalRCore_Completion *Completion)
{
  if (Completion->Error != "")
  {
    ShowMessage("Something goes wrong.");
  }
  else
  {
    ShowMessage("Invocation Successful!");
  }
}
```

Non-Blocking Invocations

The Caller sends a message to the Callee and does not expect any further messages for this invocation. Invocations can be sent without an Invocation ID value. This indicates that the invocation is "non-blocking".

Example: client invokes SendMessage method and passes as parameters user name and text message. The client doesn't expect any response from the server about the result of the invocation.

```
SignalRCore->Invoke("SendMessage", ARRAYOFCONST(("John", "Hello All.")));
```

Streaming Invocations

The Caller sends a message to the Callee and expects one or more results returned by the Callee followed by a message indicating the end of invocation.

Example: client invokes Counter method and requests 10 numbers with an interval of 500 milliseconds.

```
SignalRCore->InvokeStream("Counter", [10, 500], "id-000002");

void OnSignalRCoreStreamItem(TObject *Sender, TSignalRCore_StreamItem *StreamItem, bool &Cancel)
{
  DoLog("#stream item: " + StreamItem->Item);
}

void OnSignalRCoreCompletion(TObject *Sender, TSignalRCore_Completion *Completion)
{
  if (Completion->Error != "")
  {
    ShowMessage("Something goes wrong.");
  }
  else
  {
    ShowMessage("Invocation Successful!");
  }
}
```

Invocations

In order to perform a single invocation, the Caller follows the following basic flow:

```
void Invoke(const string aTarget, const Array of Const aArguments, const string aInvocationId)
void InvokeStream(const string aTarget, const Array of Const aArguments, const String aInvocationId)
```

Allocate a unique Invocation ID value (arbitrary string, chosen by the Caller) to represent the invocation. Call Invoke or InvokeStream method containing the Target being invoked, Arguments and InvocationId (if you don't send InvocationId, you won't get completion result).

If the Invocation is marked as non-blocking (see "Non-Blocking Invocations" below), stop here and immediately yield back to the application. Handle StreamItem or Completion message with a matching Invocation ID.

```
SignalRCore->InvokeStream("Counter", [10, 500], "id-000002");

void OnSignalRCoreStreamItem(TObject *Sender, TSignalRCore_StreamItem *StreamItem, bool &Cancel)
{
  if (StreamItem->InvocationId == "id-000002")
  {
    DoLog("#stream item: " + StreamItem->Item);
  }
}

void OnSignalRCoreCompletion(TObject *Sender, TSignalRCore_Completion *Completion)
{
  if (Completion->InvocationId == "id-000002")
  {
    if (Completion->Error != "")
    {
      ShowMessage("Something goes wrong.");
    }
    else
    {
      ShowMessage("Invocation Successful!");
    }
  }
```

```
}
```

You can call a single invocation and wait for completion.

```
bool InvokeAndWait(const String aTarget, System::TVarRec *aArguments, string aInvocationId,
  TSignalRCore_Completion &Completion, const int aTimeout = 10000);
bool InvokeStreamAndWait(const String aTarget, System::TVarRec *aArguments, string aInvocationId,
  TSignalRCore_Completion &Completion, const int aTimeout = 10000);
```

Allocate a unique Invocation ID value (arbitrary string, chosen by the Caller) to represent the invocation. Call InvokeAndWait or InvokeStreamAndWait method containing the Target being invoked, Arguments and InvocationId. The program will wait till completion event is called or Time out has been exceeded.

```
{
  TSignalRCore_Completion oCompletion;
  if (SignalRCore->InvokeStreamAndWait("Counter", ARRAYOFCONST((10, 500)), "id-000002", oCompletion))
  {
    DoLog("#invoke stream ok: " + oCompletion->Result);
  }
  else
  {
    DoLog("#invocke stream error: " + oCompletion->Error);
  }
}


void OnSignalRCoreStreamItem(TObject *Sender, TSignalRCore_StreamItem *StreamItem, bool &Cancel)
{
  if (StreamItem->InvocationId == "id-000002")
  {
    DoLog("#stream item: " + StreamItem->Item);
  }
}
```

Cancel Invocation

If the client wants to stop receiving StreamItem messages before the Server sends a Completion message, the client can send a CancelInvocation message with the same InvocationId used for the StreamInvocation message that started the stream.

```
void OnSignalRCoreStreamItem(TObject *Sender, TSignalRCore_StreamItem *StreamItem, bool &Cancel)
{
  if (StreamItem->InvocationId == "id-000002")
  {
    Cancel = true;
  }
}
```

Client Results

An Invocation is only considered completed when the Completion message is received. If the client receives an Invocation from the server, OnSignalRCoreInvocation event will be called.

```
void OnSignalRCoreInvocation(TObject *Sender, TSignalRCore_Invocation *Invocation)
{
  if (Invocation->Target == "SendMessage")
  {
    ... your code here ...
  }
}

// Once invocation is completed, call Completion method to inform server invocation is finished.
// If result is successful, then call CompletionResult method:
SignalRCore->CompletionResult("id-000002", "ok");

// If not, then call CompletionError method:
SignalRCore->CompletionError("id-000002", "Error processing invocation.");
```

Close Connection

Sent by the client when a connection is closed. Contains an error reason if the connection was closed because of an error.

```
SignalRCore->Close("Unexpected message").

// If the server close connection by any reason, OnSignalRCoreClose event will be called.
void OnSignalRCoreClose(TObject *Sender, TSignalRCore_Close *Close)
{
  DoLog("#closed: " + Close->Error);
}
```

Ping

The SignalR Hub protocol supports "Keep Alive" messages used to ensure that the underlying transport connection remains active. These messages help ensure:

Proxies don't close the underlying connection during idle times (when few messages are being sent). If the underlying connection is dropped without being terminated gracefully, the application is informed as quickly as possible.

Keep alive behaviour is achieved calling Ping method or enabling HeartBeat on WebSocket client. If the server sends a ping to the client, the client will send automatically a response and OnSignalRCoreKeepAlive event will be called.

```
void OnSignalRCoreKeepAlive(TObject *Sender)
{
  DoLog("#keepalive");
}
```

MessagePack

In the MsgPack Encoding of the SignalR Protocol, each Message is represented as a single MsgPack array containing items that correspond to properties of the given hub protocol message. The array items may be primitive values, arrays (e.g. method arguments) or objects (e.g. argument value). The first item in the array is the message type.

Refer to the MessagePack documentation to see how encode the messages sent.

Every time a new message is received, this is dispatched in the event OnSignalRCoreMessagePack event. The message can be accessed reading the Data Stream parameter. The parameter JSON by default is empty, if you convert the MessagePack message to JSON, the component will process the JSON message as if the encoding was using JSON (so the events OnSignalRCoreCompletion, OnSignalRCoreInvocation... will be dispatched).

API SignalR

SignalR

SignalR component uses WebSocket as transport to connect to a SignalR server, if this transport is not supported, an error will be raised.
SignalR client component has a property called SignalR where you can set following data:

- **Hubs:** contains a list of hubs the client is subscribing to.
- **ProtocolVersion:** the version of the protocol used by the client, supports protocol versions from 1.2 to 1.5
- **UserAgent:** user agent used to connect to SignalR server.

The client supports sending Text or Binary data.

Hubs Messages

Hubs API makes it possible to invoke server methods from the client and client methods from the server. The protocol used for persistent connection is not rich enough to allow expressing RPC (remote procedure call) semantics. It does not mean however that the protocol used for hub connections is completely different from the protocol used for persistent connections. Rather, the protocol used for hub connections is mostly an extension of the protocol for persistent connections.

When a client invokes a server method it no longer sends a free-flow string as it was for persistent connections. Instead, it sends a JSON string containing all necessary information needed to invoke the method. Here is a sample message a client would send to invoke a server method:

```
WriteData("{\"H\":\"chathub\",\"M\":\"Send\",\"A\":[\"CBuilder Client\",\"Test message\"],\"I\":0
```

The payload has the following properties:
I – invocation identifier – allows you to match up responses with requests
H – the name of the hub
M – the name of the method
A – arguments (an array, can be empty if the method does not have any parameters)

If the string argument has **double quotes** replace " by \"

Example: if the argument is {"test":1}, send the argument as {\"test\":1}

```
WriteData('{"H":"chathub","M":"Send","A":["{\"test\":1}"],"I":0}');
```

Authorization

Authentication can be enabled to associate a user with each connection and filter which users can access resources. Authentication is implemented using Bearer Tokens: the client provides an access token and the server validates this token and uses it to identify the user.

Currently only Bearer Tokens are supported:

Here, you pass token directly to Signal server (because token has been obtained from another server).

- **Authentication.Enabled:** if active, authorization will be used before a websocket connection is established.

- **Authentication.Authentication:** 2 types of authentication are supported: bearer token or cookies. Both require an external way to get the required values.
  - **BearerToken:** token value obtained.
  - **Cookie:** set the value of the cookie required.

```
TsgcWSAPI_Signal *oSignalR = new TsgcWSAPI_Signal(NULL);
oSignalR->SignalR->Enabled = true;
oSignalR->SignalR->Authentication = srcBearerToken;
oSignalR->SignalR->BearerToken->Token = "token here";
```

The component has the following events:

OnSignalRConnect

This event is raised when the client connects successfully to the server.

OnSignalRDisconnect

This event is raised when the client is disconnected from the server.

OnSignalRError

This event is called when there is an error in WebSocket connection.

OnSignalRMessage

The protocol used for persistent connection is quite simple. Messages sent to the server are just raw strings. There is no specific format they have to be in. Messages sent to the client are more structured. The properties you can find in the message are as follows:

C – message id, present for all non-KeepAlive messages
M – an array containing actual data.

```
{"C":"d-9B7A6976-B,2|C,2","M":["Welcome!"]}
```

OnSignalRBinary

This event is called when binary data is received from the server.

OnSignalRResult

When a server method is invoked, the server returns a confirmation that the invocation has completed by sending the invocation id to the client and – if the method returned a value – the return value, or – if invoking the method failed – the error.
Here are sample results of a server method call:

```
{"I":"0"}
```

A server void method whose invocation identifier was "0" completed successfully.

```
{"I":"0", "R":42}
```

A server method returning a number whose invocation identifier was "0" completed successfully and returned the value 42.

```
{"I":"0", "E":"Error occurred"}
```

OnSignalRKeepAlive

This event is raised when a KeepAlive message is received from the server.

API Kraken

Kraken

Overview

WebSockets API offers real-time market data updates. WebSockets is a bidirectional protocol offering fastest real-time data, helping you build real-time applications. The public message types presented below do not require authentication. Private-data messages can be subscribed on a separate authenticated endpoint.

Kraken offers a REST API too with Public market data and Private user data (which requires an authentication).

Configuration

Private API requires creating an API key from your Kraken account.
Kraken allows Test environment on WebSocket protocol, enable Beta property from Kraken Property to use this beta feature.

Properties

- **Kraken.Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

APIs supported

- WebSockets Public API: connects to a public WebSocket server.
- WebSockets Private API: connects to a private WebSocket server and requires an API Key and API Secret to Authenticate against server.
- REST Public API: connects to a public REST server.
- REST Private API: connects to a public REST server and requires an API Key and API Secret to Authenticate against server.

Kraken Examples

How to Connect to Public WebSocket Server

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Kraken *oKraken = new TsgcWSAPI_Kraken(NULL);
oKraken->Client = oClient;
oClient->Active = true;
```

How to Connect to Private WebSocket Server

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Kraken *oKraken = new TsgcWSAPI_Kraken(NULL);
oKraken->Kraken->ApiKey = "your api key";
oKraken->Kraken->ApiSecret = "your api secret";
oKraken->Client = oClient;
oClient->Active = true;
```

How to Get Ticker from REST API

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Kraken *oKraken = new TsgcWSAPI_Kraken(NULL);
oKraken->Client = oClient;
ShowMessage(oKraken->GetTicker(ARRAYOFCONST(("XBTUSD"))));
```

REST API Methods

Public Endpoints

Method	Arguments	Description
GetSystemStatus		Get current system status.

Private Endpoints

Method	Arguments	Description
GetExtendedBalance		Get extended balance information.
AmendOrder		Amend an existing order.
CancelAllOrders		Cancel all open orders.
CancelAllOrdersAfter		Dead man's switch - cancel all orders after timeout.
EditOrder		Edit an existing order.
AddOrderBatch		Batch add multiple orders.
CancelOrderBatch		Batch cancel multiple orders.
GetWithdrawalMethods		Get available withdrawal methods.
GetWithdrawalAddresses		Get withdrawal addresses.

How to Get Account Balance from REST API

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Kraken *oKraken = new TsgcWSAPI_Kraken(NULL);
oKraken->Kraken->ApiKey = "your api key";
oKraken->Kraken->ApiSecret = "your api secret";
oKraken->Client = oClient;
ShowMessage(oKraken->GetAccountBalance());
```

API Kraken | WebSockets Public API

Connection

URL: wss://ws.kraken.com (v1) or wss://ws.kraken.com/v2 (v2, recommended)
The component now supports WebSocket API v2 via the **Version** property (default: 2). Set `Kraken.Version := 1` to use the legacy v1 endpoint.

Once the socket is open you can subscribe to a public channel by sending a subscribe request message.

General Considerations

- All messages sent and received via WebSockets are encoded in JSON format
- All floating point fields (including timestamps) are quoted to preserve precision.
- Format of each tradeable pair is A/B, where A and B are ISO 4217-A3 for standardized assets and popular unique symbol if not standardized.
- Timestamps should not be considered unique and not be considered as aliases for transaction ids. Also, the granularity of timestamps is not representative of transaction rates.

Supported Pairs

ADA/CAD, ADA/ETH, ADA/EUR, ADA/USD, ADA/XBT, ATOM/CAD, ATOM/ETH, ATOM/EUR, ATOM/USD, ATOM/XBT, BCH/EUR, BCH/USD, BCH/XBT, DASH/EUR, DASH/USD, DASH/XBT, EOS/ETH, EOS/EUR, EOS/USD, EOS/XBT, GNO/ETH, GNO/EUR, GNO/USD, GNO/XBT, QTUM/CAD, QTUM/ETH, QTUM/EUR, QTUM/USD, QTUM/XBT, USDT/USD, ETC/ETH, ETC/XBT, ETC/EUR, ETC/USD, ETH/XBT, ETH/CAD, ETH/EUR, ETH/GBP, ETH/JPY, ETH/USD, LTC/XBT, LTC/EUR, LTC/USD, MLN/ETH, MLN/XBT, REP/ETH, REP/XBT, REP/EUR, REP/USD, STR/EUR, STR/USD, XBT/CAD, XBT/EUR, XBT/GBP, XBT/JPY, XBT/USD, BTC/CAD, BTC/EUR, BTC/GBP, BTC/JPY, BTC/USD, XDG/XBT, XLM/XBT, DOGE/XBT, STR/XBT, XLM/EUR, XLM/USD, XMR/XBT, XMR/EUR, XMR/USD, XRP/XBT, XRP/CAD, XRP/EUR, XRP/JPY, XRP/USD, ZEC/XBT, ZEC/EUR, ZEC/JPY, ZEC/USD, XTZ/CAD, XTZ/ETH, XTZ/EUR, XTZ/USD, XTZ/XBT

Methods

Ping

Client can ping server to determine whether connection is alive, server responds with pong.
This is an application level ping as opposed to default ping in WebSockets standard which is server initiated

Ticker

Ticker information includes best ask and best bid prices, 24hr volume, last trade price, volume weighted average price, etc for a given currency pair. A ticker message is published every time a trade or a group of trade happens.
Subscribe to a ticker calling SubscribeTicker method:

```
SubscribeTicker([L"XBT/USD"]);
```

If subscription is successful, **OnKrakenSubscribed** event will be called:

```
void OnKrakenSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription, string ChannelName,
  int ReqID)
{
  DoLog("#subscribed: " + Subscription + " " + Pair + " " + ChannelName);
}
```

UnSubscribe calling UnSubscribeTicker method:

```
UnSubscribeTicker(ARRAYOFCONST((L"XBT/USD")));
```

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

```
void OnKrakenUnSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#unsubscribed: " + Subscription + " " + Pair);
}
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenSubscriptionError event will be called.

```
void OnKrakenSubscriptionError(TObject *Sender, string ErrorMessage, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#subscription error: " + ErrorMessage);
}
```

Ticker updates will be notified in OnKrakenData event.

```
[
  0,
  {
    "a": [
      "5525.40000",
      1,
      "1.000"
    ],
    "b": [
      "5525.10000",
      1,
      "1.000"
    ],
    "c": [
      "5525.10000",
      "0.00398963"
    ],
    "v": [
      "2634.11501494",
      "3591.17907851"
    ],
    "p": [
      "5631.44067",
      "5653.78939"
    ],
    "t": [
      11493,
      16267
    ],
    "l": [
      "5505.00000",
      "5505.00000"
    ],
    "h": [
      "5783.00000",
      "5783.00000"
    ],
    "o": [
      "5760.70000",
      "5763.40000"
    ]
  },
  "ticker",
  "XBT/USD"
]
```

OHLC

When subscribed for OHLC, a snapshot of the last valid candle (irrespective of the endtime) will be sent, followed by updates to the running candle. For example, if a subscription is made to 1 min candle and there have been no trades for 5 mins, a snapshot of the last 1 min candle from 5 mins ago will be published. The endtime can be used to determine that it is an old candle.

Subscribe to a OHLC calling SubscribeOHLC method, you must pass pair and interval.

```
SubscribeOHLC(ARRAYOFCONST((L"XBT/USD")), kin1min);
```

If subscription is successful, OnKrakenSubscribed event will be called:

```
void OnKrakenSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription, string ChannelName,
  int ReqID)
{
  DoLog("#subscribed: " + Subscription + " " + Pair + " " + ChannelName);
}
```

UnSubscribe calling UnSubscribeOHLC method:

```
UnSubscribeOHLC(ARRAYOFCONST(("XBT/USD")), kin1min);
```

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

```
void OnKrakenUnSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#unsubscribed: " + Subscription + " " + Pair);
}
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenSubscriptionError event will be called.

```
void OnKrakenSubscriptionError(TObject *Sender, string ErrorMessage, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#subscription error: " + ErrorMessage);
}
```

OHLC updates will be notified in OnKrakenData event.

```
[
  42,
  [
    "1542057314.748456",
    "1542057360.435743",
    "3586.70000",
    "3586.70000",
    "3586.60000",
    "3586.60000",
    "3586.68894",
    "0.03373000",
    2
  ],
  "ohlc-5",
  "XBT/USD"
]
```

Trade

Trade feed for a currency pair.
Subscribe to Trade feed calling SubscribeTrade method.

```
SubscribeTrade(ARRAYOFCONST((L"XBT/USD")));
```

If subscription is successful, OnKrakenSubscribed event will be called:

```
void OnKrakenSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
  string ChannelName, int ReqID)
{
  DoLog("#subscribed: " + Subscription + " " + Pair + " " + ChannelName);
}
```

UnSubscribe calling UnSubscribeTrade method:

```
UnSubscribeTrade(ARRAYOFCONST((L"XBT/USD")));
```

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

```
void OnrakenUnSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#unsubscribed: " + Subscription + " " + Pair);
}
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenSubscriptionError event will be called.

```
void OnKrakenSubscriptionError(TObject *Sender, string ErrorMessage, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#subscription error: " + ErrorMessage);
}
```

Trade updates will be notified in OnKrakenData event.

```
[
  0,
  [
    [
      "5541.20000",
      "0.15850568",
      "1534614057.321597",
      "s",
      "l",
      ""
    ],
    [
      "6060.00000",
      "0.02455000",
      "1534614057.324998",
      "b",
      "l",
      ""
    ]
  ],
  "trade",
  "XBT/USD"
]
```

Book

Order book levels. On subscription, a snapshot will be published at the specified depth, following the snapshot, level updates will be published.
Subscribe to a Book calling SubscribeBook method, you must pass pair and depth.

```
SubscribeBook(ARRAYOFCONST((L"XBT/USD")), kde10);
```

If subscription is successful, OnKrakenSubscribed event will be called:

```
void OnKrakenSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription, string ChannelName,
  int ReqID)
{
  DoLog("#subscribed: " + Subscription + " " + Pair + " " + ChannelName);
}
```

UnSubscribe calling UnSubscribeBook method:

```
UnSubscribeBook(ARRAYOFCONST((L"XBT/USD")), kde10);
```

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

```
void OnKrakenUnSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#unsubscribed: " + Subscription + " " + Pair);
}
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenSubscriptionError event will be called.

```
void OnKrakenSubscriptionError(TObject *Sender, string ErrorMessage, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#subscription error: " + ErrorMessage);
}
```

Book updates will be notified in OnKrakenData event.

```
[
  0,
  {
    "as": [
      [
          "5541.30000",
          "2.50700000",
          "1534614248.123678"
      ],
      [
          "5541.80000",
          "0.33000000",
          "1534614098.345543"
      ],
      [
          "5542.70000",
          "0.64700000",
          "1534614244.654432"
      ]
    ],
    "bs": [
      [
          "5541.20000",
          "1.52900000",
          "1534614248.765567"
      ],
      [
          "5539.90000",
          "0.30000000",
          "1534614241.769870"
      ],
      [
          "5539.50000",
          "5.00000000",
          "1534613831.243486"
      ]
    ]
  },
  "book-100",
  "XBT/USD"
]
```

Spread

Spread feed to show best bid and ask price for subscribed asset pair. Bid volume and ask volume is part of the message too.
Subscribe to Spread feed calling SubscribeSpread method.

```
SubscribeSpread(ARRAYOFCONST(("XBT/USD")));
```

If subscription is successful, OnKrakenSubscribed event will be called:

```
void OnKrakenSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription, string ChannelName,
  int ReqID)
{
  DoLog("#subscribed: " + Subscription + " " + Pair + " " + ChannelName);
}
```

UnSubscribe calling UnSubscribeSpread method:

```
UnSubscribeSpread(ARRAYOFCONST((L"XBT/USD")));
```

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

```
void OnrakenUnSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#unsubscribed: " + Subscription + " " + Pair);
}
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenSubscriptionError event will be called.

```
void OnKrakenSubscriptionError(TObject *Sender, string ErrorMessage, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#subscription error: " + ErrorMessage);
}
```

Spread updates will be notified in OnKrakenData event.

```
[
  0,
  [
    "5698.40000",
    "5700.00000",
    "1542057299.545897",
    "1.01234567",
    "0.98765432"
  ],
  "spread",
  "XBT/USD"
]
```

Other Methods

You can subscribe / unsubscribe to all channels with one method:

```
SubscribeAll(ARRAYOFCONST((L"XBT/USD")));

UnSubscribeAll(ARRAYOFCONST((L"XBT/USD")));
```

OHLC interval value is 1 if all channels subscribed.

Events

OnConnect: when websocket client is connected to server.
OnKrakenConnect: called after successful websocket connection and when server sends system status.
OnKrakenSystemStatus: called when system status changes.
OnKrakenSubscribed: called after a successful subscription to a channel.
OnKrakenUnSubscribed: called after a successful unsubscription from a channel.
OnKrakenSubscriptionError: called if there is an error trying to subscribe / unsubscribe.
OnKrakenData: called every time a channel subscription has an update.

API Kraken | WebSockets Private API

Connection

URL: wss://ws-auth.kraken.com (v1) or wss://ws-auth.kraken.com/v2 (v2, recommended)
The component now supports WebSocket API v2 via the **Version** property (default: 2).

Once the socket is open you can subscribe to private-data channels by sending an authenticated subscribe request message.

Authentication

The API client must request an authentication "token" via the following REST API endpoint "GetWebSocketsToken" to connect to WebSockets Private endpoints. The token should be used within 15 minutes of creation. The token does not expire once a connection to a WebSockets API private message (openOrders or ownTrades) is maintained.
In order to get a Websockets Token, an API Key and API Secret must be set in Kraken Options Component, the api key provided by Kraken in your account

```
Kraken->ApiKey = "api key";
Kraken->ApiSecret = "api secret";
```

Methods

OwnTrades

Get a list of own trades, on first subscription, you get a list of latest 50 trades

```
SubscribeOwnTrades();
```

Later, you can unsubscribe from OwnTrades, calling UnSubscribeOwnTrades method

```
UnSubscribeOwnTrades();
```

Response example from server

```
[
  [
    {
      "TDLH43-DVQXD-2KHVYY": {
        "cost": "1000000.00000",
        "fee": "600.00000",
        "margin": "0.00000",
        "ordertxid": "TDLH43-DVQXD-2KHVYY",
        "ordertype": "limit",
        "pair": "XBT/EUR",
        "postxid": "OGTT3Y-C6I3P-XRI6HX",
        "price": "100000.00000",
        "time": "1560520332.914664",
        "type": "buy",
        "vol": "1000000000.00000000"
      }
    }
  ],
  "ownTrades"
]
```

Open Orders

Feed to show all the open orders belonging to the user authenticated API key. Initial snapshot will provide list of all open orders and then any updates to the open orders list will be sent. For status change updates, such as 'closed', the fields orderid and status will be present in the payload

```
SubscribeOpenOrders();
```

Later, you can unsubscribe from OpenOrders, calling UnSubscribeOpenOrders method

```
UnSubscribeOpenOrders();
```

Response example from server


```
[
  [
    {
      "OGTT3Y-C6I3P-XRI6HX": {
        "cost": "0.00000",
        "descr": {
          "close": "",
          "leverage": "0:1",
          "order": "sell 0.00001000 XBT/EUR @ limit 9.00000 with 0:1 leverage",
          "ordertype": "limit",
          "pair": "XBT/EUR",
          "price": "9.00000",
          "price2": "0.00000",
          "type": "sell"
        },
        "expiretm": "0.000000",
        "fee": "0.00000",
        "limitprice": "9.00000",
        "misc": "",
        "oflags": "fcib",
        "opentm": "0.000000",
        "price": "9.00000",
        "refid": "OKIVMP-5GVZN-Z2D2UA",
        "starttm": "0.000000",
        "status": "open",
        "stopprice": "0.000000",
        "userref": 0,
        "vol": "0.00001000",
        "vol_exec": "0.00000000"
      }
    }
  ],
  "openOrders"
]
```


Add Order

Send a new Order to Kraken

```
TsgcWSKrakenOrder *oKrakenOrder = new TsgcWSKrakenOrder();
oKrakenOrder->Pair = "XBT/USD";
oKrakenOrder->_Type = kosBuy;
oKrakenOrder->OrderType = kotMarket;
oKrakenOrder->Volume = 1;
AddOrder(oKrakenOrder);
```

List of Order parameters

```
pair = asset pair
type = type of order (buy/sell)
ordertype = order type:
    market
    limit (price = limit price)
    stop-loss (price = stop loss price)
    take-profit (price = take profit price)
    stop-loss-profit (price = stop loss price, price2 = take profit price)
    stop-loss-profit-limit (price = stop loss price, price2 = take profit price)
    stop-loss-limit (price = stop loss trigger price, price2 = triggered limit price)
```

```
    take-profit-limit (price = take profit trigger price, price2 = triggered limit price)
    trailing-stop (price = trailing stop offset)
    trailing-stop-limit (price = trailing stop offset, price2 = triggered limit offset)
    stop-loss-and-limit (price = stop loss price, price2 = limit price)
    settle-position
price = price (optional.  dependent upon ordertype)
price2 = secondary price (optional.  dependent upon ordertype)
volume = order volume in lots
leverage = amount of leverage desired (optional.  default = none)
oflags = comma delimited list of order flags (optional):
    viqc = volume in quote currency (not available for leveraged orders)
    fcib = prefer fee in base currency
    fciq = prefer fee in quote currency
    nompp = no market price protection
    post = post only order (available when ordertype = limit)
starttm = scheduled start time (optional):
    0 = now (default)
    +<n> = schedule start time <n> seconds from now
    <n> = unix timestamp of start time
expiretm = expiration time (optional):
    0 = no expiration (default)
    +<n> = expire <n> seconds from now
    <n> = unix timestamp of expiration time
userref = user reference id.  32-bit signed number.  (optional)
validate = validate inputs only.  do not submit order (optional)
optional closing order to add to system when order gets filled:
    close[ordertype] = order type
    close[price] = price
    close[price2] = secondary price
```

Response example from server

```
{
  "descr": "buy 0.01770000 XBTUSD @ limit 4000",
  "event": "addOrderStatus",
  "status": "ok",
  "txid": "ONPNXH-KMKMU-F4MR5V"
}
```

Cancel Order

Cancel order

```
CancelOrder("Order Id");
```

Response example from server

```
{
  "event": "cancelOrderStatus",
  "status": "ok"
}
```

API Kraken | REST Public API

Connection

URL: https://api.kraken.com

Kraken Public API doesn't require any authentication.

Configuration

The only configuration is whether to enable a log for REST HTTP requests. Enable HTTPLogOptions if you want to save in a text file log all HTTP Requests/Responses

Events

OnKrakenHTTPException: this event is called if there is any exception doing an HTTP Request from REST Api.

Methods

GetServerTime

This method is to aid in approximating the skew time between the server and client. Returns Time in Unix format.

```
{"error":[],"result":{"unixtime":1586705546,"rfc1123":"Sun, 12 Apr 20 15:32:26 +0000"}}
```

GetAssets

Returns information about Assets

```
{"error":[],"result":{"ADA":{"aclass":"currency","altname":"ADA","decimals":8,"display_decimals":6}}}}
```

GetAssetPairs

Returns information about a pair of assets

```
Kraken->REST_API->GetAssetPairs(ARRAYOFCONST((L"XBTUSD")));
```

GetTicker

Returns ticker information

```
Kraken->REST_API->GetTicker(ARRAYOFCONST((L"XBTUSD")));
```

GetOHLC

Returns Open-High-Low-Close data.

```
Kraken->REST_API->GetOHLC("XBTUSD");
```

GetOrderBook

Returns Array pair name and market depth.

```
Kraken->REST_API->GetOrderBook("XBTUSD");
```

GetTrades

Returns recent trade data of a pair.

```
Kraken->REST_API->GetTrades("XBTUSD");
```

GetSpread

Returns recent spread data of a pair.

```
Kraken->REST_API->GetSpread("XBTUSD");
```

API Kraken | REST Private API

Connection

URL: https://api.kraken.com

Authentication

REST Private API requires an API Key and API Secret, these values are provided by Kraken in your account.

```
Kraken->ApiKey = "api key";
Kraken->ApiSecret = "api secret";
```

Methods

GetAccountBalance

Returns your account balance.

```
Kraken->REST_API->GetAccountBalance();
```

GetTradeBalance

Returns information about your trades.

```
Kraken->REST_API->GetTradeBalance();
```

GetOpenOrders

Returns a list of open orders.

```
Kraken->REST_API->GetOpenOrders();
```

GetClosedOrders

Returns a list of closed orders.

```
Kraken->REST_API->GetClosedOrders();
```

QueryOrders

Query information about an order.

```
Kraken->REST_API->QueryOrders("1234");
```

GetTradesHistory

Returns an array of trade info.

```
Kraken->REST_API->GetTradesHistory();
```

QueryTrades

Query information about a trade.

```
Kraken->REST_API->QueryTrades("1234");
```

GetOpenPositions

Returns position info.

```
Kraken->REST_API->GetOpenPositions("1234");
```

GetLedgers

Returns associative array of ledgers info.

```
Kraken->REST_API->GetLedgers();
```

QueryLedgers

Returns associative array of ledgers info.

```
Kraken->REST_API->QueryLedgers("1234");
```

GetTradeVolume

Returns trade volume info.

```
Kraken->REST_API->GetTradeVolume();
```

AddExport

Adds a new report export.

```
Kraken->REST_API->AddExport("Report All Trades");
```

ExportStatus

Get Status of reports

```
Kraken->REST_API->ExportStatus();
```

RetrieveExport

Get Report by report id.

```
Kraken->REST_API->RetrieveExport("GOCO");
```

RemoveExport

Remove Report by report id.

```
Kraken->REST_API->RemoveExport("GOCO");
```

Add Order

Adds a new order

```
pair = asset pair
type = type of order (buy/sell)
ordertype = order type:
    market
    limit (price = limit price)
    stop-loss (price = stop loss price)
    take-profit (price = take profit price)
    stop-loss-profit (price = stop loss price, price2 = take profit price)
    stop-loss-profit-limit (price = stop loss price, price2 = take profit price)
    stop-loss-limit (price = stop loss trigger price, price2 = triggered limit price)
    take-profit-limit (price = take profit trigger price, price2 = triggered limit price)
    trailing-stop (price = trailing stop offset)
    trailing-stop-limit (price = trailing stop offset, price2 = triggered limit offset)
    stop-loss-and-limit (price = stop loss price, price2 = limit price)
    settle-position
price = price (optional.  dependent upon ordertype)
price2 = secondary price (optional.  dependent upon ordertype)
volume = order volume in lots
leverage = amount of leverage desired (optional.  default = none)
oflags = comma delimited list of order flags (optional):
    viqc = volume in quote currency (not available for leveraged orders)
    fcib = prefer fee in base currency
    fciq = prefer fee in quote currency
    nompp = no market price protection
    post = post only order (available when ordertype = limit)
starttm = scheduled start time (optional):
    0 = now (default)
    +n = schedule start time n seconds from now
    n = unix timestamp of start time
expiretm = expiration time (optional):
    0 = no expiration (default)
    +n = expire n seconds from now
    n = unix timestamp of expiration time
userref = user reference id.  32-bit signed number.  (optional)
validate = validate inputs only.  do not submit order (optional)
optional closing order to add to system when order gets filled:
    close[ordertype] = order type
    close[price] = price
    close[price2] = secondary price
```

```
TsgcHTTPKrakenOrder *oKrakenOrder = new TsgcHTTPKrakenOrder();
oKrakenOrder->Pair = "XBT/USD";
oKrakenOrder->_Type = koshBuy;
oKrakenOrder->OrderType = kothMarket;
oKrakenOrder->Volume = 1;
Kraken->REST_API->AddOrder(oKrakenOrder);
```

CancelOrder

Cancels an open order by id

```
Kraken->REST_API->CancelOrder("1234");
```

API Kraken v2

Kraken

Overview

The **TsgcWSAPI_Kraken_V2** component implements the Kraken **WebSocket API v2**. It offers the same real-time market data and account data as the v1 API, but over a different wire protocol, with a JSON object per message instead of the v1 JSON array, and with named fields instead of positional ones.
The public channels do not require authentication. The private channels (executions and balances) and the trading methods require an authentication token, which the component requests from the Kraken REST API using the API Key and the API Secret configured in the component.

Why a separate component

The v2 API is **not** a new version of the v1 messages, it is a different schema, so it is implemented as a separate component and the v1 component **TsgcWSAPI_Kraken** is left untouched.

- v2 has **no channelID**. A subscription is identified by the channel name and the symbol, so the **ChannelId** parameter carried by the v1 **OnKrakenSubscribed** and **OnKrakenUnSubscribed** events has no v2 equivalent, and the v1 **UnSubscribe(aChannelId)** method has no meaning in v2 either.
- v1 frames are keyed on **event** / **pair** / **subscription** and the payloads arrive as JSON arrays. v2 frames are keyed on **method** / **params** and the payloads on **channel** / **type**, always as JSON objects.
- Because the two components are independent, **existing applications built on TsgcWSAPI_Kraken are not affected**. Nothing has been renamed, deprecated or removed in the v1 component, and both components can be used at the same time in the same application.

Frame shape

Every request written by the component is a JSON object with a **method**, an optional **params** object and a **req_id**:

```
{
  "method": "subscribe",
  "params": {
    "channel": "ticker",
    "symbol": [
      "BTC/USD"
    ]
  },
  "req_id": 1
}
```

The **req_id** is an incremental counter maintained by the component, it correlates every acknowledgement with the request that produced it and it is reported back in the **ReqID** parameter of the subscription events. It does not have to be supplied by the application.
The payloads arrive keyed on the channel and the message type:

```
{
  "channel": "ticker",
  "type": "snapshot",
  "data": [
    {
      "symbol": "BTC/USD",
      "bid": 65120.1,
      "bid_qty": 0.5,
      "ask": 65120.9,
      "ask_qty": 1.2,
      "last": 65120.4,
      "volume": 1352.09,
      "change": 210.5,
      "change_pct": 0.32
    }
  ]
}
```

Connection

The endpoint is selected automatically. When the **Kraken.ApiKey** is not empty, the component connects to the authenticated endpoint, so the public and the private channels share a single connection.

Configuration	URL
Public	wss://ws.kraken.com/v2
Private (ApiKey set)	wss://ws-auth.kraken.com/v2
Public + Beta	wss://beta-ws.kraken.com/v2
Private + Beta	wss://beta-ws-auth.kraken.com/v2

The component enables the WebSocket heartbeat with a 30 second interval before connecting, unless the application has already enabled it with its own interval.

Configuration

The API Key is generated exactly the same way as for the v1 component, from your Kraken account. That procedure is not repeated here, see the API Kraken topic, and then copy the key and the secret to the **Kraken** property of the component.

Properties

- **Kraken.ApiKey:** the API key created in your Kraken account. Required by the private channels and by the trading methods. When it is not empty the component connects to the authenticated endpoint.
- **Kraken.ApiSecret:** the API secret, used to sign the REST request that mints the WebSocket token.
- **Kraken.Beta:** if enabled, the component connects to the Kraken beta hosts instead of the production ones (by default false).
- **Kraken.Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false). The private channels are never replayed as they were sent, the frame is rebuilt with a freshly minted token.
- **Kraken.HTTPLogOptions:** enables a log file for the REST requests made by the component, which is where the WebSocket token is requested.
- **REST_API:** the Kraken REST client used internally. See the REST API section below.

Symbols

v2 uses the common ticker where v1 uses the Kraken asset code. The component translates the symbols automatically, so **XBT/USD** is written on the wire as **BTC/USD** and **XDG/USD** as **DOGE/USD**. Both spellings can be passed to the subscribe methods and to the trading methods. The same translation is available to the application calling the **GetSymbolV2** method.

REST API

The v2 API is a **WebSocket-only** API, there is no v2 REST API. The component reuses the very same Kraken REST client the v1 component uses, exposed through the **REST_API** property, so the public and private REST endpoints are documented once, in the v1 pages:

- REST Public API: connects to a public REST server.
- REST Private API: connects to a public REST server and requires an API Key and API Secret to Authenticate against server.

APIs supported

- WebSockets Public API: the ticker, book, ohlc, trade and instrument channels, no authentication required, plus level3, which is market data but requires an API Key and an API Secret because Kraken requires an authentication token on its subscribe request.
- WebSockets Private API: the executions and balances channels and the trading methods. Requires an API Key and an API Secret.

Kraken v2 Examples

How to Connect to Public WebSocket Server

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Kraken_V2 *oKraken = new TsgcWSAPI_Kraken_V2(NULL);
oKraken->Client = oClient;
oClient->Active = true;
```

How to Connect to Private WebSocket Server

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Kraken_V2 *oKraken = new TsgcWSAPI_Kraken_V2(NULL);
oKraken->Kraken->ApiKey = "your api key";
oKraken->Kraken->ApiSecret = "your api secret";
oKraken->Client = oClient;
oClient->Active = true;
```

How to Subscribe to the Ticker Channel

```
void OnKrakenV2Connect(TObject *Sender, String ConnectionId, String Status, String ApiVersion,
  String Version)
{
  oKraken->SubscribeTicker(ARRAYOFCONST((L"BTC/USD", L"ETH/USD")));
}

void OnKrakenV2Data(TObject *Sender, String Channel, String MessageType, String aData)
{
  DoLog("#data: " + Channel + " " + MessageType + " " + aData);
}
```

Events

OnConnect: when websocket client is connected to server.
OnKrakenV2Connect: called when the first status message of the session arrives, it reports the connection id, the system status, the api version and the version of the API.
OnKrakenV2Status: called on every following status message, when the status of the API changes.
OnKrakenV2Subscribed: called after a successful subscription to a channel, it reports the channel, the symbol and the req_id of the request.
OnKrakenV2UnSubscribed: called after a successful unsubscription from a channel.
OnKrakenV2SubscriptionError: called if there is an error trying to subscribe / unsubscribe.
OnKrakenV2Data: called every time a channel subscription has an update, it reports the channel, the message type (snapshot, update...) and the raw payload.
OnKrakenV2Heartbeat: called every time the server sends a heartbeat message.
OnKrakenData: called with the raw text of any message which is not one of the above, for example the acknowledgements of the trading methods. It is also called with the payload of a channel update when no OnKrakenV2Data handler is assigned, so no message is ever discarded.
OnKrakenHTTPException: called if there is an exception in the REST request that mints the WebSocket token.
OnDisconnect: when websocket client is disconnected from server.

API Kraken v2 | WebSockets Public API

Connection

URL: wss://ws.kraken.com/v2
If the **Kraken.Beta** property is enabled, the component connects to wss://beta-ws.kraken.com/v2 instead. If an **Kraken.ApiKey** is configured, the component connects to the authenticated endpoint and the public channels listed here are subscribed on that same connection.
Once the socket is open you can subscribe to a public channel calling one of the Subscribe methods, none of them requires an API Key.

General Considerations

- All messages sent and received via WebSockets are encoded in JSON format.
- Unlike v1, every v2 message is a JSON **object** and the fields are named, there are no positional arrays and there is no channelID.
- Prices, quantities and timestamps are JSON values, prices and quantities are numbers and timestamps are RFC3339 strings.
- A subscription is identified by the channel name plus the symbol, which is what the **OnKrakenV2Subscribed** and **OnKrakenV2UnSubscribed** events report.
- Each request carries an incremental **req_id** minted by the component, which is reported back in the ReqID parameter of the subscription events.
- The format of each tradeable symbol is A/B, using the common ticker. The component translates the Kraken asset codes automatically, so XBT/USD is sent as BTC/USD and XDG/USD as DOGE/USD.

Methods

Ping

The client can ping the server to determine whether the connection is alive, the server responds with a pong. This is an application level ping as opposed to the default ping of the WebSockets standard, which is server initiated.

```
Ping();
```

The frame written on the wire is:

```
{
  "method": "ping",
  "req_id": 1
}
```

Ticker

Ticker information includes the best bid and ask prices with their quantities, the last traded price, the 24 hour volume, the volume weighted average price and the change over the last 24 hours, for a given symbol.
Subscribe to the ticker channel calling the SubscribeTicker method:

```
SubscribeTicker(ARRAYOFCONST((L"BTC/USD")));
```

If the subscription is successful, **OnKrakenV2Subscribed** event will be called:

```
void OnKrakenV2Subscribed(TObject *Sender, String Channel, String Symbol, int ReqID)
{
  DoLog("#subscribed: " + Channel + " " + Symbol);
}
```

UnSubscribe calling the UnSubscribeTicker method:

```
UnSubscribeTicker(ARRAYOFCONST((L"BTC/USD")));
```

If the unsubscription is successful, OnKrakenV2UnSubscribed event will be called:

```
void OnKrakenV2UnSubscribed(TObject *Sender, String Channel, String Symbol, int ReqID)
{
  DoLog("#unsubscribed: " + Channel + " " + Symbol);
}
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenV2SubscriptionError event will be called.

```
void OnKrakenV2SubscriptionError(TObject *Sender, String ErrorMessage, String Channel, String Symbol,
  int ReqID)
{
  DoLog("#subscription error: " + ErrorMessage);
}
```

Ticker updates will be notified in the OnKrakenV2Data event, with the Channel parameter set to "ticker".

```
{
  "channel": "ticker",
  "type": "update",
  "data": [
    {
      "symbol": "BTC/USD",
      "bid": 65120.1,
      "bid_qty": 0.50000000,
      "ask": 65120.9,
      "ask_qty": 1.20000000,
      "last": 65120.4,
      "volume": 1352.09384756,
      "vwap": 64980.2,
      "low": 64105.0,
      "high": 65890.0,
      "change": 210.5,
      "change_pct": 0.32
    }
  ]
}
```

Book

Aggregated order book levels. On subscription a snapshot is published at the requested depth, and after the snapshot the level updates are published. Pass the symbols and the depth.

```
SubscribeBook(ARRAYOFCONST((L"BTC/USD")), kde10);

UnSubscribeBook(ARRAYOFCONST((L"BTC/USD")), kde10);
```

The supported depths are the values of the **TsgcWSKrakenDepth** enumerated type: **kde10** (10), **kde25** (25), **kde100** (100), **kde500** (500) and **kde1000** (1000).
Book updates will be notified in the OnKrakenV2Data event, with the Channel parameter set to "book".

```
{
  "channel": "book",
  "type": "snapshot",
  "data": [
    {
      "symbol": "BTC/USD",
      "bids": [
        {
          "price": 65120.1,
          "qty": 0.50000000
        },
        {
          "price": 65119.8,
          "qty": 1.75000000
        }
      ],
```

```
      "asks": [
        {
          "price": 65120.9,
          "qty": 1.20000000
        },
        {
          "price": 65121.4,
          "qty": 0.33000000
        }
      ],
      "checksum": 3251967404
    }
  ]
}
```

Level3

The individual orders of the book instead of the aggregated price levels, so every order is published with its own order id, price, quantity and timestamp. This channel only exists in v2, it has no v1 counterpart. Although the data itself is market data, Kraken requires the subscribe request to carry an authentication token, the same way it does for the Executions and Balances channels. **SubscribeLevel3** and **UnSubscribeLevel3** mint and attach that token automatically, so configure **Kraken.ApiKey** and **Kraken.ApiSecret** before connecting, subscribing with no credentials configured fails.
Level3 accepts a depth of 10, 100 or 1000 only. The depth is passed with the same **TsgcWSKrakenDepth** enumeration the Book channel uses, so **kde25** and **kde500** raise an exception and nothing is sent.

```
SubscribeLevel3(ARRAYOFCONST((L"BTC/USD")), kde10);

UnSubscribeLevel3(ARRAYOFCONST((L"BTC/USD")), kde10);
```

Level3 updates will be notified in the OnKrakenV2Data event, with the Channel parameter set to "level3".

```
{
  "channel": "level3",
  "type": "update",
  "data": [
    {
      "symbol": "BTC/USD",
      "bids": [
        {
          "order_id": "OZDPFP-QCVYP-C5KHFC",
          "limit_price": 65120.1,
          "order_qty": 0.50000000,
          "timestamp": "2026-08-20T09:12:44.123456Z",
          "event": "add"
        }
      ],
      "asks": [],
      "checksum": 1974322581
    }
  ]
}
```

OHLC

When subscribed to OHLC, a snapshot of the last valid candle is sent, followed by the updates of the running candle. Pass the symbols and the interval.

```
SubscribeOHLC(ARRAYOFCONST((L"BTC/USD")), kin1min);

UnSubscribeOHLC(ARRAYOFCONST((L"BTC/USD")), kin1min);
```

The supported intervals are the values of the **TsgcWSKrakenInterval** enumerated type: **kin1min**, **kin5min**, **kin15min**, **kin30min**, **kin60min**, **kin240min**, **kin1440min**, **kin10080min** and **kin21600min**, written on the wire as the interval in minutes.
OHLC updates will be notified in the OnKrakenV2Data event, with the Channel parameter set to "ohlc".

```
{
  "channel": "ohlc",
```

```
  "type": "update",
  "timestamp": "2026-08-20T09:13:00.000000Z",
  "data": [
    {
      "symbol": "BTC/USD",
      "open": 65118.0,
      "high": 65131.2,
      "low": 65110.5,
      "close": 65120.4,
      "vwap": 65121.3,
      "trades": 42,
      "volume": 3.45872100,
      "interval_begin": "2026-08-20T09:12:00.000000000Z",
      "interval": 1
    }
  ]
}
```

Trade

Trade feed for a symbol, every executed trade is published with its price, quantity, side, order type and timestamp.

```
SubscribeTrade(ARRAYOFCONST((L"BTC/USD")));

UnSubscribeTrade(ARRAYOFCONST((L"BTC/USD")));
```

Trade updates will be notified in the OnKrakenV2Data event, with the Channel parameter set to "trade".

```
{
  "channel": "trade",
  "type": "update",
  "data": [
    {
      "symbol": "BTC/USD",
      "side": "buy",
      "price": 65120.4,
      "qty": 0.01770000,
      "ord_type": "limit",
      "trade_id": 74123845,
      "timestamp": "2026-08-20T09:12:44.654321Z"
    }
  ]
}
```

Instrument

The reference data of the exchange, the list of assets and the list of tradeable pairs with their precision, their minimum order size and their status. This channel takes no symbol, it publishes the whole catalogue as a snapshot and then the changes.

```
SubscribeInstrument();

UnSubscribeInstrument();
```

Instrument updates will be notified in the OnKrakenV2Data event, with the Channel parameter set to "instrument".

```
{
  "channel": "instrument",
  "type": "snapshot",
  "data": {
    "assets": [
      {
        "id": "BTC",
        "status": "enabled",
        "precision": 10,
        "precision_display": 5,
        "borrowable": true
      }
    ],
    "pairs": [
      {
        "symbol": "BTC/USD",
```

```
        "base": "BTC",
        "quote": "USD",
        "status": "online",
        "qty_precision": 8,
        "qty_min": 0.00005000,
        "price_precision": 1,
        "cost_precision": 5
      }
    ]
  }
}
```

Heartbeat

The server publishes a heartbeat message about once per second while the connection is idle. It is not subscribed, it is always published, and it is reported in the **OnKrakenV2Heartbeat** event instead of OnKrakenV2Data.

```
void OnKrakenV2Heartbeat(TObject *Sender)
{
  DoLog("#heartbeat");
}
```

Resubscribe

If the **Kraken.Resubscribe** property is enabled, the channels that were active before a disconnection are subscribed again automatically after the reconnection. The frames are rebuilt and a new req_id is minted for each one, so the ReqID reported by OnKrakenV2Subscribed after a reconnection is not the one of the original request.

Events

OnConnect: when websocket client is connected to server.
OnKrakenV2Connect: called when the first status message of the session arrives.
OnKrakenV2Status: called when the status of the API changes.
OnKrakenV2Subscribed: called after a successful subscription to a channel.
OnKrakenV2UnSubscribed: called after a successful unsubscription from a channel.
OnKrakenV2SubscriptionError: called if there is an error trying to subscribe / unsubscribe.
OnKrakenV2Data: called every time a channel subscription has an update.
OnKrakenV2Heartbeat: called every time the server sends a heartbeat message.
OnKrakenData: called with the raw text of any message which is not dispatched by the events above.

API Kraken v2 | WebSockets Private API

Connection

URL: wss://ws-auth.kraken.com/v2
The endpoint is selected automatically: as soon as the **Kraken.ApiKey** property is not empty, the component connects to the authenticated endpoint, and the public channels are subscribed on the very same connection.

Authentication

The API client must request an authentication "token" via the REST API endpoint **GetWebSocketsToken** to subscribe to the private channels and to send the trading methods. This is the very same mechanism the v1 API uses, and the token is requested with the very same REST client, so no extra configuration is needed. The token should be used within 15 minutes of creation, and it does not expire while the connection that uses it stays open.
In order to get a WebSockets token, an API Key and an API Secret must be set in the Kraken options of the component, the API key provided by Kraken in your account.

```
Kraken->ApiKey = "api key";
Kraken->ApiSecret = "api secret";
```

The token is requested on demand, the first time a private channel or a trading method is called, and it is cached for the rest of the session. The application never handles it.
Difference with v1. The token itself is minted exactly the same way, what changed is how it travels: v1 writes it at the root of the frame, next to the event name, while v2 carries it **inside the params object** of every private frame, next to the channel and the parameters of the request:

```
{
  "method": "subscribe",
  "params": {
    "channel": "executions",
    "snapshot": true,
    "token": "your websockets token"
  },
  "req_id": 1
}
```

If **Kraken.Resubscribe** is enabled, the private subscriptions are never replayed as they were sent, since the token they carried belongs to the session that has just been lost. The cached token is dropped on reconnection and each private frame is rebuilt with a freshly minted one.

Beta hosts

Kraken offers a beta environment for the WebSocket protocol. Enable the **Kraken.Beta** property of the component to use it, and the component connects to **wss://beta-ws-auth.kraken.com/v2** instead of the production host (or to wss://beta-ws.kraken.com/v2 when no API Key is configured). The property must be set before the connection is opened.

```
Kraken->Beta = true;
```

Channels

Executions

The executions channel replaces the v1 **ownTrades** and **openOrders** channels with a single feed: it reports the whole life cycle of the orders of the authenticated account, the new orders, the amendments, the fills, the cancellations and the expirations, each one with its own execution type.

```
SubscribeExecutions(true);

UnSubscribeExecutions();
```

Pass **aSnapshot** as False to skip the initial snapshot of the open orders and receive only the updates that happen from then on.
Executions updates will be notified in the OnKrakenV2Data event, with the Channel parameter set to "executions".

```
{
  "channel": "executions",
  "type": "update",
  "data": [
    {
      "order_id": "OZDPFP-QCVYP-C5KHFC",
      "exec_type": "filled",
      "exec_id": "TDLH43-DVQXD-2KHVYY",
      "symbol": "BTC/USD",
      "side": "buy",
      "order_type": "limit",
      "limit_price": 65000.0,
      "order_qty": 0.01770000,
      "cum_qty": 0.01770000,
      "order_status": "filled",
      "timestamp": "2026-08-20T09:12:44.654321Z"
    }
  ]
}
```

Balances

The balances channel reports the balance of every asset of the authenticated account, and then every change of those balances, with the ledger entry that caused it.

```
SubscribeBalances(true);

UnSubscribeBalances();
```

Balances updates will be notified in the OnKrakenV2Data event, with the Channel parameter set to "balances".

```
{
  "channel": "balances",
  "type": "snapshot",
  "data": [
    {
      "asset": "BTC",
      "balance": 1.25470000,
      "wallets": [
        {
          "type": "spot",
          "id": "main",
          "balance": 1.25470000
        }
      ]
    },
    {
      "asset": "USD",
      "balance": 15320.4500,
      "wallets": [
        {
          "type": "spot",
          "id": "main",
          "balance": 15320.4500
        }
      ]
    }
  ]
}
```

Trading

The trading methods are written on the same connection as the channels, each one as a v2 frame with its own method name, its params and a req_id, and each one carries the authentication token. The acknowledgement sent

back by Kraken is reported in the **OnKrakenData** event, and the resulting order state arrives in the executions channel.

Method	Frame	Description
AddOrder	add_order	Sends a new order. Takes a TsgcWSKrakenV2Order object with the parameters of the order.
AmendOrder	amend_order	Changes the quantity or the price of an open order keeping its order id. Takes the order id and, optionally, the new order quantity, limit price, trigger price and display quantity. Only the values that are not empty are sent.
EditOrder	edit_order	Replaces an open order with a new one, which gets a new order id. Takes the order id and the symbol, and optionally the new order quantity, limit price and trigger price.
CancelOrder	cancel_order	Cancels one or more orders by order id, the ids are passed as an open array.
CancelAll	cancel_all	Cancels every open order of the account.
CancelAllOrdersAfter	cancel_all_orders_after	Dead man's switch. Cancels every open order after the given timeout in seconds, unless the method is called again before it elapses. A timeout of 0 disarms it.
BatchAdd	batch_add	Sends several orders in a single frame. The symbol is a batch level parameter, so every order of the batch belongs to the same symbol and the Symbol property of each order is ignored. Kraken accepts from 2 to 15 orders per batch.
BatchCancel	batch_cancel	Cancels several orders in a single frame, the ids are passed as an open array. Each element is a Kraken order id (string) or an order_userref (integer). Kraken accepts from 2 to 50 ids per batch.

TsgcWSKrakenV2Order

The parameters of an order are set in a **TsgcWSKrakenV2Order** object, which is the v2 counterpart of the v1 TsgcWSKrakenOrder class. It is a different class because the v2 parameters are named differently and the composite v1 order types do not exist in v2.

Property	Field	Description
Symbol	symbol	The traded pair, for example BTC/USD. The Kraken asset codes are translated automatically, XBT becomes BTC and XDG becomes DOGE. Ignored by BatchAdd, which takes the symbol as a batch level parameter.
Side	side	kosBuy or kosSell, written as buy or sell.
OrderType	order_type	The type of the order, see the table below.
OrderQty	order_qty	The quantity of the order, in units of the base asset.
LimitPrice	limit_price	The limit price, required by the limit order types.
TriggerPrice	triggers.price	The price that triggers the order, required by the stop loss and take profit types. It is written inside the triggers object of the frame.
DisplayQty	display_qty	The visible quantity of an iceberg order, the rest of the quantity stays hidden.
TimeInForce	time_in_force	How long the order stays on the book: gtc, gtd or ioc.
OrderUserRef	order_userref	A user reference id of your own, a 32-bit signed number, echoed back in the executions channel.

PostOnly	post_only	If True, the order is rejected instead of executed when it would take liquidity from the book.
ReduceOnly	reduce_only	If True, the order can only reduce an open position, never increase it.
Validate	validate	If True, the order is only validated by the server and never submitted. Useful to check the parameters of an order before trading with it.

The money fields (OrderQty, LimitPrice, TriggerPrice and DisplayQty) are **strings** and not floating point numbers on purpose, so the exact decimal representation supplied by the application is written on the wire, without rounding and without the decimal separator of the locale. They are written as JSON numbers, so pass them unquoted and with a dot as the decimal separator.

Order types

The **OrderType** property takes a value of the **TsgcWSKrakenOrderType** enumerated type, which is shared with the v1 component. v2 spells the order types with a dash where v1 spells them with an underscore, and the composite v1 types (kotStopLossProfit, kotStopLossProfitLimit and kotStopLossAndLimit) do not exist in v2.

```
kotMarket            = market
kotLimit             = limit
kotStopLoss          = stop-loss
kotTakeProfit        = take-profit
kotStopLossLimit     = stop-loss-limit
kotTakeProfitLimit   = take-profit-limit
kotTrailingStop      = trailing-stop
kotTrailingStopLimit = trailing-stop-limit
```

Add Order

Send a new order to Kraken.

```
TsgcWSKrakenV2Order *oOrder = new TsgcWSKrakenV2Order();
try
{
  oOrder->Symbol = "BTC/USD";
  oOrder->Side = kosBuy;
  oOrder->OrderType = kotLimit;
  oOrder->OrderQty = "0.01770000";
  oOrder->LimitPrice = "65000.0";
  oOrder->TimeInForce = "gtc";
  oOrder->PostOnly = true;
  AddOrder(oOrder);
}
__finally
{
  delete oOrder;
}
```

The frame written on the wire is:

```
{
  "method": "add_order",
  "params": {
    "order_type": "limit",
    "side": "buy",
    "symbol": "BTC/USD",
    "order_qty": 0.01770000,
    "limit_price": 65000.0,
    "time_in_force": "gtc",
    "post_only": true,
    "token": "your websockets token"
  },
  "req_id": 2
}
```

Response example from server, reported in the OnKrakenData event:

```
{
  "method": "add_order",
  "req_id": 2,
  "result": {
    "order_id": "OZDPFP-QCVYP-C5KHFC"
  },
  "success": true,
  "time_in": "2026-08-20T09:12:44.123456Z",
  "time_out": "2026-08-20T09:12:44.128901Z"
}
```


Amend Order

Change the quantity or the price of an open order without losing its place in the queue and without changing its order id. Only the arguments that are not empty are written on the frame.

```
AmendOrder("OZDPFP-QCVYP-C5KHFC", "0.02500000", "64900.0");
```

Edit Order

Replace an open order with a new one. Unlike AmendOrder, the order is cancelled and resubmitted, so it gets a new order id and it goes to the back of the queue. The symbol is required.

```
EditOrder("OZDPFP-QCVYP-C5KHFC", "BTC/USD", "0.02500000", "64900.0");
```

Cancel Order

Cancel one or more orders by order id.

```
CancelOrder(ARRAYOFCONST((L"OZDPFP-QCVYP-C5KHFC")));
CancelOrder(ARRAYOFCONST((L"OZDPFP-QCVYP-C5KHFC", L"OGTT3Y-C6I3P-XRI6HX")));
```

Cancel All

Cancel every open order of the account.

```
CancelAll();
```

Cancel All Orders After

Dead man's switch. Every open order is cancelled once the given timeout elapses, unless the method is called again before that, which is how an application tells Kraken it is still alive. Call it with a timeout of 0 to disarm it.

```
CancelAllOrdersAfter(60);
```

Batch Add

Send several orders in a single frame. The symbol is passed once, as a batch level parameter, and it is not repeated on every order of the batch.
Kraken validates the whole batch at once, so **validate** is a batch level parameter too. When any order of the batch has **Validate** set to True, the batch is sent with **validate** set to true and none of its orders reaches the matching engine.
Kraken accepts from 2 to 15 orders per batch and rejects the whole batch when the count is out of that range, so **BatchAdd** raises an exception and sends nothing in that case.

```
TsgcWSKrakenV2Order *oOrders[2];
oOrders[0] = new TsgcWSKrakenV2Order();
oOrders[1] = new TsgcWSKrakenV2Order();
try
{
  oOrders[0]->Side = kosBuy;
  oOrders[0]->OrderType = kotLimit;
  oOrders[0]->OrderQty = "0.01000000";
  oOrders[0]->LimitPrice = "64000.0";

  oOrders[1]->Side = kosBuy;
  oOrders[1]->OrderType = kotLimit;
  oOrders[1]->OrderQty = "0.01000000";
  oOrders[1]->LimitPrice = "63500.0";

  BatchAdd("BTC/USD", oOrders, 1);
}
__finally
{
  delete oOrders[0];
  delete oOrders[1];
}
```

The frame written on the wire is:

```
{
  "method": "batch_add",
  "params": {
    "symbol": "BTC/USD",
    "orders": [
      {
        "order_type": "limit",
        "side": "buy",
        "order_qty": 0.01000000,
        "limit_price": 64000.0
      },
      {
        "order_type": "limit",
        "side": "buy",
        "order_qty": 0.01000000,
        "limit_price": 63500.0
      }
    ],
    "token": "your websockets token"
  },
  "req_id": 3
}
```

Batch Cancel

Cancel several orders in a single frame. Each element of the array is either a Kraken order id, passed as a string, or an order_userref, passed as an integer.
Kraken accepts from 2 to 50 ids per batch and rejects the whole batch when the count is out of that range, so **BatchCancel** raises an exception and sends nothing in that case.

```
BatchCancel(ARRAYOFCONST((L"OZDPFP-QCVYP-C5KHFC", L"OGTT3Y-C6I3P-XRI6HX")));
```

Events

OnKrakenV2Subscribed: called after a successful subscription to the executions or the balances channel, the Channel parameter reports which one.
OnKrakenV2UnSubscribed: called after a successful unsubscription.
OnKrakenV2SubscriptionError: called if the subscription is rejected, for example when the token is invalid or has expired.
OnKrakenV2Data: called with the payloads of the executions and balances channels.
OnKrakenData: called with the acknowledgements of the trading methods.
OnKrakenHTTPException: called if there is an exception in the REST request that mints the WebSocket token, for example when the API Key or the API Secret are not valid.

API Kraken Futures

Kraken Futures

Overview

The **REST API** allows you to securely access the methods of your Kraken Futures account. Examples of REST API Methods:

- request current or historical price information
- check your account balance and PnL
- your margin parameters and estimated liquidation thresholds
- place or cancel orders (individually or in batch)
- see your open orders
- open positions or trade history
- request a digital asset withdrawal

These methods are called "endpoints" and are explained in REST API section.

The **Websocket API** allows you to securely establish a communication channel to the Kraken Futures platform to receive information in real time. This allows listening to updates instead of continuously sending requests. These channels are called subscriptions.

Some of the endpoints allow performing sensitive tasks, such as initiating a digital asset withdrawal. To access these endpoints securely, the API uses encryption techniques developed by the National Security Agency.

Configuration

In order to use the API, you need to generate a pair of unique **API keys** (if you want access to private APIs):

1. Sign in to your **Kraken Futures account**.
2. Click on your name on the upper-right corner.
3. Select "Settings" from the drop-down menu.
4. Select the "Create Key" tab in the API panel.
5. Press the "Create Key" button.
6. View your Public and Private keys and record them somewhere safe.

Copy the Public and Private Keys to the **KrakenOptions** property of the component.

KrakenOptions.ApiKey
KrakenOptions.ApiSecret
KrakenOptions.Resubscribe: if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

APIs supported

- WebSockets Public API: connects to a public WebSocket server.
- WebSockets Private API: connects to a private WebSocket server and requires an API Key and API Secret to Authenticate against server.
- REST Public API: connects to a public REST server.
- REST Private API: connects to a public REST server and requires an API Key and API Secret to Authenticate against server.

REST API Methods

Private Endpoints

Method	Arguments	Description
BatchOrder		Submit batch orders (place/cancel/edit).
GetOrderStatus		Get status of specific orders.
GetPNLCurrencyPreferences		Get PNL currency preferences.
SetPNLCurrencyPreference		Set PNL currency preference for a symbol.
GetLeverageSettings		Get leverage preferences.
SetLeverageSettings		Set leverage for a symbol.

API Kraken Futures | WebSockets Public API

Connection

URL: wss://futures.kraken.com/ws/v1

Once the socket is open you can subscribe to a public channel by sending a subscribe request message.

Methods

Ticker

This endpoint returns current market data for all currently listed Futures contracts and indices. Authentication is not required.
Subscribe to a ticker calling **SubscribeTicker** method:

```
SubscribeTicker(ARRAYOFCONST(("PI_XBTUSD")));
```

If subscription is successful, **OnKrakenFuturesSubscribed** event will be called:

```
void OnKrakenFuturesSubscribed(TObject *Sender, string Feed, string ProductId)
{
  DoLog("#subscribed: " + Feed + " " + ProductId);
}
```

UnSubscribe calling **UnSubscribeTicker** method:

```
UnSubscribeTicker(ARRAYOFCONST(("PI_XBTUSD")));
```

If unsubscription is successful, **OnKrakenFuturesUnSubscribed** event will be called:

```
void OnKrakenFuturesUnSubscribed(TObject *Sender, string Feed, string ProductId)
{
  DoLog("#unsubscribed: " + Feed + " " + ProductId);
}
```

If there is an error while trying to subscribe / unsubscribe, **OnKrakenFuturesError** event will be called.

```
void OnKrakenFuturesError(TObject *Sender, string Error)
{
  DoLog("#error: " + Error);
}
```

Ticker updates will be notified in **OnKrakenData** event.

```
{    "result":"success",

 "tickers":[

{

"tag": "perpetual",

"pair": "XBT:USD",
```

"symbol": "pi_xbtusd",

"markPrice": 9520.2,

"bid": 9520,

"bidSize": 30950,

"ask": 9520.5,

"askSize": 3779,

"vol24h": 68238712,

"openInterest": 29308193,

"open24h": 10137,

"last": 9521,

"lastTime": "2020-06-03T08:14:26.624Z",

"lastSize": 1,

"suspended": false,

"fundingRate": 4.943012455e-9,

"fundingRatePrediction": 4.414499215e-9

}

{

"tag": "quarter",

"pair": "XBT:USD",

"symbol": "fi_xbtusd_200925",

"markPrice": 9659.8,

"bid": 9659.5,

"bidSize": 6480,

"ask": 9660,

"askSize": 17100,

"vol24h": 4562580,

"openInterest": 3573325,

"open24h": 10370.5,

"last": 9660,

"lastTime": "2020-06-03T08:10:37.800Z",

"lastSize": 5000,

"suspended": false

},

{

"symbol": "in_xbtusd",

"last": 9519,

"lastTime": "2020-06-03T08:14:49.000Z"

}

],

"serverTime": "2020-06-03T08:14:49.865Z"

}

Trade

The trade feed returns information about executed trades
Subscribe to Trade feed calling **SubscribeTrade** method.

```
SubscribeTrade(ARRAYOFCONST(("PI_XBTUSD")));
```

If subscription is successful, **OnKrakenFuturesSubscribed** event will be called:

```
void OnKrakenFuturesSubscribed(TObject *Sender, string Feed, string ProductId,)
{
  DoLog("#subscribed: " + Feed + " " + ProductId);
}
```

UnSubscribe calling **UnSubscribeTrade** method:

```
UnSubscribeTrade(ARRAYOFCONST(("PI_XBTUSD")));
```

If unsubscription is successful, **OnKrakenFuturesUnSubscribed** event will be called:

```
void OnrakenFuturesUnSubscribed(TObject *Sender, string Feed, string ProductId)
{
  DoLog("#unsubscribed: " + Feed + " " + ProductId);
}
```

If there is an error while trying to subscribe / unsubscribe, **OnKrakenFuturesError** event will be called.

```
void OnKrakenFuturesError(TObject *Sender, string Error)
{
  DoLog("#error: " + Error);
}
```

Trade updates will be notified in **OnKrakenData** event.

{ "feed": "trade_snapshot",

 "product_id": "PI_XBTUSD",

 "trades": [

```
{
  "feed": "trade",
  "product_id": "PI_XBTUSD",
  "uid": "caa9c653-420b-4c24-a9f1-462a054d86f1",
  "side": "sell",
  "type": "fill",
  "seq": 655508,
  "time": 1612269657781,
  "qty": 440,
  "price": 34893
},
{
  "feed": "trade",
  "product_id": "PI_XBTUSD",
  "uid": "45ee9737-1877-4682-bc68-e4ef818ef88a",
  "side": "sell",
  "type": "fill",
  "seq": 655507,
  "time": 1612269656839,
  "qty": 9643,
  "price": 34891
}
 ]
}
```


Book

This feed returns information about the order book.
Subscribe to a Book calling SubscribeBook method, you must pass the Symbol.

```
SubscribeBook(ARRAYOFCONST(("PI_XBTUSD")));
```

If subscription is successful, **OnKrakenFuturesSubscribed** event will be called:

```
void OnKrakenFuturesSubscribed(TObject *Sender, string Feed, string ProductId)
{
  DoLog("#subscribed: " + Feed + " " + ProductId);
}
```

UnSubscribe calling **UnSubscribeBook** method:

```
UnSubscribeBook(ARRAYOFCONST(("PI_XBTUSD")));
```

If unsubscription is successful, **OnKrakenFuturesUnSubscribed** event will be called:

```
void OnKrakenFuturesUnSubscribed(TObject *Sender, string Feed, string ProductId)
{
  DoLog("#unsubscribed: " + Feed + " " + ProductId);
}
```

If there is an error while trying to subscribe / unsubscribe, **OnKrakenFuturesError** event will be called.

```
void OnKrakenFuturesError(TObject *Sender, string Error)
{
  DoLog("#error: " + Error);
}
```

Book updates will be notified in **OnKrakenData** event.

{

"feed": "book_snapshot",

"product_id": "PI_XBTUSD",

"timestamp": 1612269825817,

"seq": 326072249,

"tickSize": null,

"bids": [{

"price": 34892.5,

"qty": 6385

},

{

"price": 34892,

"qty": 10924

}

],

"asks": [{

"price": 34911.5,

"qty": 20598

},

{

"price": 34912,

"qty": 2300

}

]

}

Ticker Lite

The ticker lite feed returns ticker information about listed products.
Subscribe to Spread feed calling **SubscribeTickerLite** method.

```
SubscribeTickerLite(ARRAYOFCONST(("PI_XBTUSD")));
```

If subscription is successful, **OnKrakenFuturesSubscribed** event will be called:

```
void OnKrakenFuturesSubscribed(TObject *Sender, string Feed, string ProductId)
{
  DoLog("#subscribed: " + Feed + " " + ProductId);
}
```

UnSubscribe calling **UnSubscribeTickerLite** method:

```
UnSubscribeTickerLite(ARRAYOFCONST(("PI_XBTUSD")));
```

If unsubscription is successful, **OnKrakenFuturesUnSubscribed** event will be called:

```
void OnKrakenFuturesUnSubscribed(TObject *Sender, string Feed, string ProductId)
{
  DoLog("#unsubscribed: " + Feed + " " + ProductId);
}
```

If there is an error while trying to subscribe / unsubscribe, **OnKrakenFuturesError** event will be called.

```
void OnKrakenFuturesError(TObject *Sender, string Error)
{
  DoLog("#error: " + Error);
}
```

Spread updates will be notified in **OnKrakenData** event.


```
{  "feed": "ticker_lite",

  "product_id": "PI_XBTUSD",

  "bid": 34932,

  "ask": 34949.5,

  "change": 3.3705205220015966,

  "premium": 0.1,

  "volume": 264126741,

  "tag": "perpetual",

  "pair": "XBT:USD",

  "dtm": 0,
```

"maturityTime": 0

}

{

"feed":"ticker_lite",

"product_id":"FI_ETHUSD_210625",

"bid":1753.45,

"ask":1760.35,

"change":13.448175559936647,

"premium":9.1,

"volume":6899673.0,

"tag":"semiannual",

"pair":"ETH:USD",

"dtm":141,

"maturityTime":1624633200000

}

HeartBeat

The heartbeat feed publishes a heartbeat message at timed intervals.

```
SubscribeHeartBeat();

UnSubscribeHeartBeat();
```

Events

OnConnect: when websocket client is connected to server.
OnKrakenFuturesConnect: called after successful websocket connection and when server sends system status.
OnKrakenFuturesSubscribed: called after a successful subscription to a channel.
OnKrakenFuturesUnSubscribed: called after a successful unsubscription from a channel.
OnKrakenFuturesError: called if there is any error while subscribing/unsubscribing.
OnKrakenData: called every time a channel subscription has an update.

API Kraken Futures | WebSockets Private API

Connection

URL: wss://futures.kraken.com/ws/v1

Authentication

The subscribe and unsubscribe requests to WebSocket private feeds require a signed challenge message with the user api_secret.
The challenge is obtained as is shown in Section WebSocket API Public (using the api_key).
Authenticated requests must include both the original challenge message (original_challenge) and the signed (signed_challenge) in JSON format.
In order to get a Websockets Challenge, an API Key and API Secret must be set in Kraken Options Component, the api key provided by Kraken in your account

```
Kraken->ApiKey = "api key";
Kraken->ApiSecret = "api secret";
```

Methods

Open Orders Verbose

This subscription feed publishes information about user open orders. This feed adds extra information about all the post-only orders that failed to cross the book.

```
SubscribeOpenOrdersVerbose();
```

Later, you can unsubscribe from OpenOrdersVerbose, calling **UnSubscribeOpenOrdersVerbose** method

```
UnSubscribeOpenOrdersVerbose();
```

Response example from server

```
{        'feed':'open_orders_verbose_snapshot',    'account':'0f9c23b8-63e2-40e4-9592-6d5aa57c12
        {           'instrument':'PI_XBTUSD',         'time':1567428848005,         'last_updat
```

Open Positions

This subscription feed publishes the open positions of the user account.

```
SubscribeOpenPositions();
```

Later, you can unsubscribe from OpenPositions, calling **UnSubscribeOpenPositions** method

```
UnSubscribeOpenPositions();
```

Response example from server

```
{
    "feed": "open_positions",
    "account": "DemoUser",
    "positions": [{
        "instrument": "fi_xbtusd_180316",
        "balance": 2000.0,
        "entry_price": 11675.86541981,
        "mark_price": 11090.0,
        "index_price": 12290.550000000001,
        "pnl": -0.00905299
    }]
}
```

Account Log

This subscription feed publishes account information.

```
SubscribeAccountLog();
```

Later, you can unsubscribe from AccountLog, calling **UnSubscribeAccountLog** method

```
UnSubscribeAccountLog();
```

Response example from server

```
{
    'feed': 'account_log_snapshot',
    'logs': [{
                'id': 1690,
                'date': '2019-07-11T08:00:00.000Z',
                'asset': 'bch',
                'info': 'funding
                rate change ','
                booking_uid ':'
                86 fdc252 - 1 b6e - 40 ec - ac1d - c7bd46ddeebf ','
                margin_account ':'
                f - bch: usd ','
                old_balance ':0.01215667051,'
                new_balance ':0.01215736653,'
                old_average_entry_price ':0.0,'
                new_average_entry_price ':0.0,'
                trade_price ':0.0,'
                mark_price ':0.0,'
                realized_pnl ':0.0,'
                fee ':0.0,'
                execution ':'
                ','
                collateral ':'
                bch ','
                funding_rate ':-8.7002552653e-08,'
                realized_funding ':6.9602e-07}]
}
```

Fills

This subscription feed publishes fills information.

```
SubscribeFills();
```

Later, you can unsubscribe from Fills, calling **UnSubscribeFills** method

```
UnSubscribeFills();
```

Response example from server

{

"feed":"fills_snapshot",

"account":"DemoUser",

"fills":[

{

"instrument":"FI_XBTUSD_200925",

"time":1600256910739,

"price":10937.5,

"seq":36,

"buy":true,

"qty":5000.0,

"order_id":"9e30258b-5a98-4002-968a-5b0e149bcfbf",

"fill_id":"cad76f07-814e-4dc6-8478-7867407b6bff",

"fill_type":"maker",

"fee_paid":-0.00009142857,

"fee_currency":"BTC"

}]

}

Open Orders

This subscription feed publishes information about user open orders.

```
SubscribeOpenOrders();
```

Later, you can unsubscribe from OpenOrders, calling **UnSubscribeOpenOrders** method

```
UnSubscribeOpenOrders();
```

Response example from server

{

"feed": "open_orders_snapshot",

"account": "e258dba9-4dd4-4da5-bfef-75beb91c098e",

"orders": [

{

"instrument": "PI_XBTUSD",

"time": 1612275024153,

"last_update_time": 1612275024153,

"qty": 1000,

"filled": 0,

"limit_price": 34900,

"stop_price": 13789,

"type": "stop",

"order_id": "723ba95f-13b7-418b-8fcf-ab7ba6620555",

"direction": 1,

"reduce_only": false,

"triggerSignal": "last"

}

]

}

Account Balance And Margins

This subscription feed returns balance and margin information for the client's account.

```
SubscribeAccountBalanceAndMargins();
```

Later, you can unsubscribe from AccounBalance, calling **UnSubscribeAccountBalanceAndMargins** method

```
UnSubscribeAccountBalanceAndMargins();
```

Response example from server

{

"feed": "account_balances_and_margins",

"account": "DemoUser",

"margin_accounts": [

{

"name": "xbt",
"balance": 0,
"pnl": 0,
"funding": 0,
"pv": 0,
"am": 0,
"im": 0,
"mm": 0
},
{
"name": "f-xbt:usd",
"balance": 9.99730211055,
"pnl": -0.00006034858674327812,
"funding": 0,
"pv": 9.997241761963258,
"am": 9.99666885201038,
"im": 0.0005729099528781564,
"mm": 0.0002864549764390782
},
],
"seq": 14
}

Notifications

This subscription feed publishes notifications to the client.

```
SubscribeNotifications();
```

Later, you can unsubscribe from Notifications, calling **UnSubscribeNotifications** method

```
UnSubscribeNotifications();
```

Response example from server

{
"feed":"notifications_auth",

```
  "notifications":[
    {
      "id":5,
      "type":"market",
      "priority":"low",
      "note":"A note describing the notification.",
      "effective_time":1520288300000
    },
    ...
  ]
}
```

API Kraken Futures | REST Public API

Connection

URL: https://futures.kraken.com/derivatives/api/v3

Kraken Futures Public API doesn't require any authentication.

Configuration

The only configuration is whether to enable a log for REST HTTP requests. Enable HTTPLogOptions if you want to save in a text file log all HTTP Requests/Responses

Events

OnKrakenHTTPException: this event is called if there is any exception doing an HTTP Request from REST Api.

Methods

GetFeeSchedules

This endpoint lists all fee schedules. Authentication is not required.

```
KrakenFutures->REST_API->GetFeeSchedules();
```

Order Book

This endpoint returns the entire non-cumulative order book of currently listed Futures contracts.

```
KrakenFutures->REST_API->GetOrderBook("PI_XBTUSD");
```

Tickers

This endpoint returns current market data for all currently listed Futures contracts and indices.

```
KrakenFutures->REST_API->GetTickers();
```

Instruments

This endpoint returns specifications for all currently listed Futures contracts and indices.

```
KrakenFutures->REST_API->GetInstruments();
```

History

This endpoint returns the last 100 trades from the specified lastTime value - if no value specified will return the last 100 trades. is endpoint only returns trade history for a maximum of 7 days from the time it is called or since last .trading engine release (whichever is sooner).

```
KrakenFutures->REST_API->GetHistory("PI_XBTUSD");
```

API Kraken Futures | REST Private API

Connection

URL: https://futures.kraken.com/derivatives/api/v3

Authentication

REST Private API requires an API Key and API Secret, these values are provided by Kraken in your account.

```
Kraken->ApiKey = "api key";
Kraken->ApiSecret = "api secret";
```

Methods

EditOrderByOrderId

This endpoint allows editing an existing order for a currently listed Futures contract.

aOrderId: ID of the order you wish to edit
aSize: The size associated with the order
aLimitPrice: The limit price associated with the order.
aStopPrice: The stop price associated with a stop order. Required if old Order Type is Stop.

```
KrakenFutures->REST_API->EditOrderByOrderId("Order_Id", 2, 1000);
```

EditOrderByCliOrderId

This endpoint allows editing an existing order for a currently listed Futures contract.

aCliOrderId: The order identity that is specified from the user. It must be globally unique.
aSize: The size associated with the order
aLimitPrice: The limit price associated with the order.
aStopPrice: The stop price associated with a stop order. Required if Order Type is Stop.

```
KrakenFutures->REST_API->EditOrderByCliOrderId("Cli_Order_Id", 2, 1000);
```

SendMarketOrder

This endpoint allows you to send a Market Order.

aSide: The direction of the order: buy or sell.
aSymbol: The symbol of the futures
aSize: The size associated with the order.

```
KrakenFutures->REST_API->SendMarketOrder(kosfBuy, "PI_XBTUSD", 1);
```

SendLimitOrder

This endpoint allows you to send a Limit Order.

aSide: The direction of the order: buy or sell.
aSymbol: The symbol of the futures
aSize: The size associated with the order.
aLimitPrice: The limit price associated with the order.

```
KrakenFutures->REST_API->SendLimitOrder(kosfBuy, "PI_XBTUSD", 1, 1000);
```

SendStopOrder

This endpoint allows you to send a Stop Order.

aSide: The direction of the order: buy or sell.
aSymbol: The symbol of the futures
aSize: The size associated with the order.
aStopPrice: The stop price associated with a stop order.
aLimitPrice: The limit price associated with the order.

```
KrakenFutures->REST_API->SendStopOrder(kosfBuy, "PI_XBTUSD", 1, 1000, 900);
```

SendTakeProfitOrder

This endpoint allows you to send a Take Profit Order.

aSide: The direction of the order: buy or sell.
aSymbol: The symbol of the futures
aSize: The size associated with the order.
aStopPrice: The stop price associated with a stop order.
aLimitPrice: The limit price associated with the order.

```
KrakenFutures->REST_API->SendTakeProfitOrder(kosfBuy, "PI_XBTUSD", 1, 1000, 900);
```

SendOrder

This endpoint allows sending a limit, stop, take profit or immediate-or-cancel order for a currently listed Futures contract.

OrderType: select one of the following kotfLMT, kotfPOST, kotfMKT, kotfSTP, kotfTAKE_PROFIT, kotfIOC
Symbol: The symbol of the futures
Side: The direction of the order (buy or sell).
Size: The size associated with the order.
StopPrice: The stop price associated with a stop order.
LimitPrice: The limit price associated with the order.
TriggerSignal: If placing a Stop or TakeProfit order, the signal used for trigger, select one of the following kots-Mark, kotsIndex, kotsLast
CliOrderId: The order identity that is specified from the user. It must be globally unique.
ReduceOnly: Set as true if you wish the order to only reduce an existing position. Any order which increases an existing position will be rejected. Default false.

```
TsgcHTTPKrakenFuturesOrder *oOrder = new TsgcHTTPKrakenFuturesOrder(this);
try
{
  oOrder->Side = kosfBuy;
  oOrder->Symbol = "PI_XBTUSD";
  oOrder->OrderType = kotfMKT;
  oOrder->Size = 1;
  KrakenFutures->REST_API->SendOrder(oOrder);
```

```
}
__finally
{
  oOrder->Free;
}
```

CancelOrderByOrderId

This endpoint allows cancelling an open order for a Futures contract.

aOrderId: ID of the order you wish to edit

```
KrakenFutures->REST_API->CancelOrderByOrderId("Order_Id");
```

CancelOrderByCliOrderId

This endpoint allows cancelling an open order for a Futures contract.

aCliOrderId: The order identity that is specified from the user. It must be globally unique.

```
KrakenFutures->REST_API->CancelOrderByCliOrderId("Cli_Order_Id");
```

GetFills

This endpoint returns information on filled orders for all futures contracts.

aLastFillDate: If not provided, returns the last 100 fills in any futures contract. If provided, returns the 100 entries before lastFillTime.

```
KrakenFutures->REST_API->GetFills("2020-07-22T13:45:00.000Z");
```

Transfer

This endpoint allows you to transfer funds between two margin accounts with the same collateral currency, or between a margin account and your cash account.

aFromAccount: The name of the cash or margin account to move funds from.
aToAccount: The name of the cash or margin account to move funds to.
aUnit: The unit to transfer.
aAmount: The amount to transfer.

```
KrakenFutures->REST_API->Transfer("FI_XBTUSD", "cash", "xbt", 1.5);
```

GetOpenPositions

This endpoint returns the size and average entry price of all open positions in Futures contracts. This includes Futures contracts that have matured but have not yet been settled.

```
KrakenFutures->REST_API->GetOpenPositions();
```

GetNotifications

This endpoint provides the platform's notifications.

```
KrakenFutures->REST_API->GetNotifications();
```

GetAccounts

This endpoint returns key information relating to all your Kraken Futures accounts which may either be cash accounts or margin accounts. This includes digital asset balances, instrument balances, margin requirements, margin trigger estimates and auxiliary information such as available funds, PnL of open positions and portfolio value.

```
KrakenFutures->REST_API->GetAccounts();
```

CancelAllOrders

This endpoint allows cancelling an open order for a Futures contract.

Symbol: A futures product to cancel all open orders (optional)

```
KrakenFutures->REST_API->CancelAllOrders();
```

CancelAllOrdersAfter

This endpoint provides a Dead Man's Switch mechanism to protect the client from network malfunctions. The client can send a request with a timeout in seconds which will trigger a countdown timer that will cancel all client orders when timeout expires.

aTimeout: The timeout specified in seconds.

```
KrakenFutures->REST_API->CancelAllOrdersAfter(60);
```

GetOpenOrders

This endpoint returns information on all open orders for all Futures contracts.

```
KrakenFutures->REST_API->OpenOrders();
```

GetHistoricalOrders

This endpoint returns historical orders made on an account.

aSince: The DateTime Since
aBefore: The DateTime Before
aSort: "asc" for ascending sort "desc" for descending
aContinuationToken: Continuation token provided from a prior response which can be used in call to return the next set of available results

```
KrakenFutures->REST_API->GetHistoricalOrders(Now, Now - 5);
```

GetHistoricalTriggers

This endpoint returns allows historical triggers made on an account.

aSince: The DateTime Since
aBefore: The DateTime Before
aSort: "asc" for ascending sort "desc" for descending

aContinuationToken: Continuation token provided from a prior response which can be used in call to return the next set of available results

```
KrakenFutures->REST_API->GetHistoricalTriggers(Now, Now - 5);
```

GetHistoricalExecutions

This endpoint returns allows historical executions made on an account.

aSince: The DateTime Since
aBefore: The DateTime Before
aSort: "asc" for ascending sort "desc" for descending
aContinuationToken: Continuation token provided from a prior response which can be used in call to return the next set of available results

```
KrakenFutures->REST_API->GetHistoricalExecutions(Now, Now - 5);
```

WithdrawalToSpotWallet

This endpoint allows submitting a request to withdraw digital assets from a Kraken Futures wallet to your Kraken Spot wallet.

aCurrency: The digital asset that shall be withdrawn, e.g. xbt or xrp.
aAmount: The amount of currency that shall be withdrawn.

```
KrakenFutures->REST_API->WithdrawalToSpotWallet("xbt", 1000);
```

GetFeeScheduleVolumes

This endpoint returns your 30-day USD volume.

```
KrakenFutures->REST_API->GetFeeScheduleVolumes();
```

GetAccountLogCSV

This endpoint allows clients to download a csv file of their account logs.

```
KrakenFutures->REST_API->GetAccountLogCSV();
```

API Pusher

Pusher

Pusher is an easy and reliable platform with nice features based on WebSocket protocol: flexible pub/sub messaging, live user lists (presence), authentication...
Pusher WebSocket API is 7.
Data is sent bi-directionally over a WebSocket as text data containing UTF8 encoded JSON (Binary WebSocket frames are not supported).
You can call **Ping** method to test connection to the server. Essentially any messages received from the other party are considered to mean that the connection is alive. In the absence of any messages, either party may check that the other side is responding by sending a ping message, to which the other party should respond with a pong.
Before you connect, you must complete the following fields:

```
Pusher->Cluster = "eu"; // cluster where your pusher account is located
Pusher->Key = "9c3b7ef25qe97a00116c"; // your pusher api key
Pusher->Name = "js"; // optional, name of your application
Pusher->Version = "4.1";  // optional, version of your application
Pusher->TLS = true; // if encrypted, set to true
Pusher->Secret = "2dc792e1916ac49e6b3f"; // pusher secret string (needed for private and presence channels)
```

Important

Pusher requires that websocket client connects to a URL using previous fields (key, cluster...), these fields are used to build the url and this is done when you assign the client in pusher component. So, to be sure that URL is built correctly, set the client after you have filled the pusher configuration fields. Find below pseudo-code:

```
// configure pusher fields
pusher.cluster = ...
pusher.key = ...
// set client
pusher.client = websocket client
// start connection
websocket client.Active = true;
```

After a successful connection, **OnPusherConnect** event is raised and you get following fields:

- Socket ID: A unique identifier for the connected client.
- Timeout: The number of seconds of server inactivity after which the client should initiate a ping message (this is handled automatically by component).

In case of error, **OnPusherError** will be raised, and information about error provided. An error may be sent from Pusher in response to invalid authentication, an invalid command, etc.

4000-4099

Indicates an error resulting in the connection being closed by Pusher, and that attempting to reconnect using the same parameters will not succeed.

4000: Application only accepts SSL connections, reconnect using wss://
4001: Application does not exist
4003: Application disabled
4004: Application is over connection quota
4005: Path not found
4006: Invalid version string format
4007: Unsupported protocol version
4008: No protocol version supplied

4100-4199

Indicates an error resulting in the connection being closed by Pusher, and that the client may reconnect after 1s or more.

4100: Over capacity

4200-4299

Indicates an error resulting in the connection being closed by Pusher, and that the client may reconnect immediately.

4200: Generic reconnect immediately
4201: Pong reply not received: ping was sent to the client, but no reply was received - see ping and pong messages
4202: Closed after inactivity: The client has been inactive for a long time (currently 24 hours) and client does not support ping. Please upgrade to a newer WebSocket draft or implement version 5 or above of this protocol.

4300-4399

Any other type of error.
4301: Client event rejected due to rate limit

Channels

Channels are a fundamental concept in Pusher. Each application has a number of channels, and each client can choose which channels it subscribes to.
Channels provide:

- A way of filtering data. For example, in a chat application, there may be a channel for people who want to discuss 'dogs'
- A way of controlling access to different streams of information. For example, a project management application would want to authorise people to get updates about 'projectX'

It's strongly recommended that channels are used to filter your data and that it is not achieved using events. This is because all events published to a channel are sent to all subscribers, regardless of their event binding.
Channels don't need to be explicitly created and are instantiated on client demand. This means that creating a channel is easy. Just tell a client to subscribe to it.
The following types of channels are supported:

- **Public channels** can be subscribed to by anyone who knows their name
- **Private channels** introduce a mechanism which lets your server control access to the data you are broadcasting
- **Presence channels** are an extension of private channels. They let you register user information on subscription, and let other members of the channel know who's online
- **Cache channels** remember the last triggered event and send it as the first event to new subscribers (public, private and presence variants)
- **Private-Encrypted channels** provide end-to-end encryption using NaCl secretbox, ensuring that even Pusher cannot read the message data

Public Channels

Public channels should be used for publicly accessible data as they do not require any form of authorisation in order to be subscribed to.
You can subscribe and unsubscribe from channels at any time. There's no need to wait for the Pusher to finish connecting first.
Example: subscribe to channel "my-channel".

```
CBuilder
APIPusher->Subscribe("my-channel");
```

If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.
All messages from the subscribed channel will be received **OnPusherEvent** event.
When Publish method is called and the channel is Public, the component instead of using the WebSocket protocol, uses the HTTP protocol and calls the method TriggerEvent (publish is not allowed using websocket protocol).

Private Channels

Requires Indy 10.5.7 or later
Private channels should be used when access to the channel needs to be restricted in some way. In order for a user to subscribe to a private channel permission must be authorised.
Example: subscribe to channel "my-private-channel".

```
CBuilder
APIPusher->Subscribe("my-private-channel", pscPrivateChannel);
```

If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.
All messages from the subscribed channel will be received **OnPusherEvent** event.

Presence Channels

Requires Indy 10.5.7 or later
Presence channels build on the security of Private channels and expose the additional feature of an awareness of who is subscribed to that channel. This makes it extremely easy to build chat room and "who's online" type functionality to your application. Think chat rooms, collaborators on a document, people viewing the same web page, competitors in a game, that kind of thing.
Presence channels are subscribed to from the client API in the same way as private channels but the channel name must be prefixed with presence-. As with private channels an HTTP Request is made to a configurable authentication URL to determine if the current user has permissions to access the channel.
Information on users subscribing to, and unsubscribing from a channel can then be accessed by binding to events on the presence channel and the current state of users subscribed to the channel is available via the channel.members property.
Example: subscribe to channel "my-presence-channel".

```
APIPusher->Subscribe("my-presence-channel", pscPresenceChannel,
  '{"\"user_id\":"\"John_Smith"\","\"user_info\":{"\"name\":"\"John Smith"\"}}");
```

If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.
All messages from the subscribed channel will be received **OnPusherEvent** event.

Cache Channels

A cache channel remembers the last triggered event, and sends this as the first event to new subscribers.
When an event is triggered on a cache channel, Pusher Channels caches this event, and when a client subscribes to a cache channel, if a cached value exists, this is sent to the client as the first event on that channel. This behavior helps developers to provide the initial state without adding additional logic to fetch it from elsewhere.
The following Cache Channels are supported:

- Public Cache Channel
- Private Cache Channel
- Presence Cache Channel

Example: subscribe to public cache channel "my-cache-channel".

```
APIPusher->Subscribe("my-cache-channel", pscCacheChannel);
```

If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.
All messages from the subscribed channel will be received **OnPusherEvent** event.
If there is no cached event when subscribing to a cache channel, the **OnPusherCacheMiss** event will be raised, providing the channel name. This allows your application to handle the case where no cached data is available.

Private-Encrypted Channels

Private-Encrypted channels provide end-to-end encryption for messages. Like private channels, they require authentication, but additionally all data payloads are encrypted using NaCl secretbox so that only authorized subscribers can read the content. Even Pusher itself cannot decrypt the messages.
To use private-encrypted channels, you must provide a **SharedSecret** during authentication. The shared secret is used for encrypting and decrypting message data.
Example: subscribe to a private-encrypted channel "my-encrypted-channel".

```
APIPusher->Subscribe("my-encrypted-channel", pscPrivateEncryptedChannel);
```

A private-encrypted-cache variant is also available, combining encryption with cache channel behavior:

```
APIPusher->Subscribe("my-encrypted-cache-channel", pscPrivateEncryptedCacheChannel);
```

When using the **OnPusherAuthentication** event with private-encrypted channels, you can set the **SharedSecret** property on the response object to provide the encryption key:

```
void OnPusherAuthenticationEvent(TObject *Sender,
  TsgcWSPusherRequestAuthentication *AuthRequest,
  TsgcWSPusherResponseAuthentication *AuthResponse)
{
  AuthResponse->SharedSecret = "your-shared-secret-key";
}
```

Presence Events

Presence channels provide additional events that notify your application when users join or leave a channel, and allow you to track subscription counts.

OnPusherMemberAdded

Raised when a new member subscribes to a presence channel. Provides the channel name, user ID, and user info of the member that joined.

```
void PUSHERPusherMemberAdded(TObject *Sender,
  String Channel, String UserId, String UserInfo)
{
  Log("Member joined: " + UserId + " on " + Channel);
}
```

OnPusherMemberRemoved

Raised when a member unsubscribes from a presence channel. Provides the channel name, user ID, and user info of the member that left.

```
void PUSHERPusherMemberRemoved(TObject *Sender,
  String Channel, String UserId, String UserInfo)
{
  Log("Member left: " + UserId + " on " + Channel);
}
```

OnPusherSubscriptionCount

Raised when the subscription count changes on a channel. Provides the channel name and the current number of subscribers. This event must be enabled on your Pusher dashboard.

```
void PUSHERPusherSubscriptionCount(TObject *Sender,
  String Channel, int SubscriptionCount)
{
  Log(Channel + " has " + IntToStr(SubscriptionCount) + " subscribers");
}
```

OnPusherCacheMiss

Raised when subscribing to a cache channel that has no cached event. Provides the channel name. This allows your application to handle the case when no cached data is available, for example by fetching the data from another source.

```
void PUSHERPusherCacheMiss(TObject *Sender, String Channel)
{
  Log("Cache miss on: " + Channel);
}
```

Publish Messages

Not only you can receive messages from subscribed channels, but you can also send messages to other subscribed users.

Call method **Publish** to send a message to all subscribed users of channel.
Example: send an event to all subscribed users of "my-channel"

```
APIPusher->Publish("my-event", "my-channel");
```

Publish no more than 10 messages per second per client (connection). Any events triggered above this rate limit will be rejected by Pusher API. This is not a system issue, it is a client issue. 100 clients in a channel sending messages at this rate would each also have to be processing 1,000 messages per second! Whilst some modern browsers might be able to handle this it's most probably not a good idea.

REST API

The API is hosted at http://api-CLUSTER.pusher.com , where CLUSTER is replaced with your own apps cluster (for instance, eu).
HTTP status codes are used to indicate the success or otherwise of requests. The following status are common:
200 Successful request. Body will contain a JSON hash of response data
400 Error: details in response body
401 Authentication error: response body will contain an explanation
403 Forbidden: app disabled or over message quota
The following REST API functions have been implemented.

Function	Description
TriggerEvent	Triggers a new event on the specified channel. Supports optional SocketId (to exclude a client) and Info parameters.
TriggerBatchEvents	Triggers multiple events in a single HTTP request. Accepts a JSON array of event objects.
GetChannels	Provides a list of all active channels. Supports optional FilterByPrefix and Info parameters.
GetChannel	Provides information about a specific channel. Supports an optional Info parameter.
GetUsers	Provides a list of all users connected to a channel.
TerminateUserConnections	Terminates all connections for a given user by their user ID.

TriggerEvent

Triggers an event on one or more channels. Requires the event name, channel name, and data payload.

Parameter	Description
aEventName	The name of the event to trigger.
aChannel	The channel name to trigger the event on.
aData	The event data (JSON string).
aSocketId (optional)	A socket ID to exclude from receiving the event. Useful to prevent the sender from receiving its own message.
aInfo (optional)	A comma-separated list of attributes to include in the response (e.g. "subscription_count").

```
// trigger event on a channel
```

```
APIPusher->TriggerEvent("my-event", "my-channel", "Hello World");

// trigger event excluding the sender
APIPusher->TriggerEvent("my-event", "my-channel", "Hello World", "123.456");

// trigger event requesting subscription_count in the response
APIPusher->TriggerEvent("my-event", "my-channel", "Hello World", "", "subscription_count");
```

TriggerBatchEvents

Triggers multiple events in a single API call, which is more efficient than making separate requests for each event. The batch parameter must be a JSON string containing an array of event objects, where each object has "channel", "name", and "data" fields.

```
APIPusher->TriggerBatchEvents(
  "[{\"channel\":\"my-channel\",\"name\":\"my-event\",\"data\":\"hello\"},"
  "{\"channel\":\"my-channel-2\",\"name\":\"my-event\",\"data\":\"world\"}]");
```

GetChannels

Returns a list of active channels. Supports optional parameters to filter the results and request additional information.

Parameter	Description
aFilterByPrefix (optional)	Filter channels by a name prefix (e.g. "presence-" to list only presence channels).
aInfo (optional)	A comma-separated list of attributes to include in the response (e.g. "user_count").

```
// get all channels
APIPusher->GetChannels();

// get only presence channels with user count
APIPusher->GetChannels("presence-", "user_count");
```

GetChannel

Returns information about a specific channel.

Parameter	Description
aChannel	The channel name to get information about.
aInfo (optional)	A comma-separated list of attributes to include (e.g. "user_count,subscription_count").

```
// get channel info
APIPusher->GetChannel("presence-my-channel");

// get channel info with user count
APIPusher->GetChannel("presence-my-channel", "user_count,subscription_count");
```

GetUsers

Returns a list of users connected to a presence channel. The channel name must include the full prefix (e.g. "presence-my-channel").

```
APIPusher->GetUsers("presence-my-channel");
```

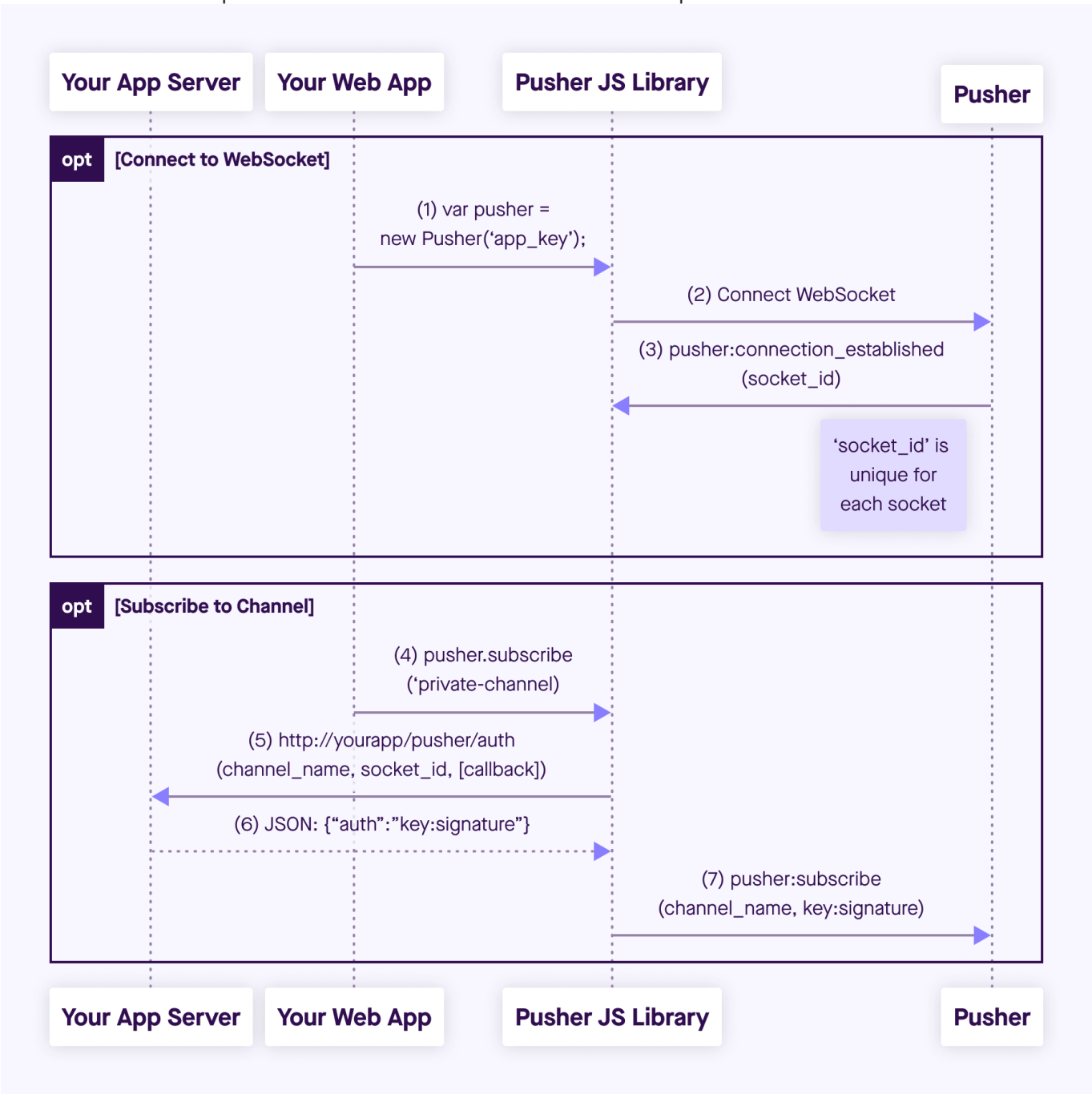
TerminateUserConnections

Terminates all connections established by a given user. This can be used to force a specific user to disconnect from all channels. The user ID must match the "user_id" used when the user subscribed to a presence channel.

```
APIPusher->TerminateUserConnections("1234");
```

Custom Authentication

Pusher only allows subscribing to private or presence channels, if the connection provides an authentication token, this allows you to restrict the access.
You can build your own Authentication flow, using **OnPusherAuthentication** event, this event is called before the subscription message is signed with the secret key provided by Pusher. This event has 2 parameters a request authentication with fields like SocketId, channel name... which can be used by your own authentication server to authenticate or not the request. Find below a screenshot which shows the pusher authentication flow



When a client connects to the pusher server, it sends the Key provided by pusher and the server returns an identification id (socket_id).

When a client subscribes to a private (or presence) channel, the sgcWebSockets client uses the Secret Key provided by pusher to create a signature which is included in the subscription message. Using the OnPusherAutentication event, you can capture the fields required to sign the message, implement your own authentication methods and if successful, return the signature and this signature will be included in the subscription message and sent to the server.
Example:

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Pusher *oPusher = new TsgcWSAPI_Pusher(NULL);
oPusher->Client = oClient;
oPusher->Cluster = "eu";
Pusher->Name = "js";
Pusher->Version = "4.1";
Pusher->TLS = true;
Pusher->Key = "9c3b7ef25qe97a00116c";
Pusher->Secret = ""; // the secret key is not known by the client, only by the authentication module

oPusher->OnPusherAuthentication = OnPusherAuthenticationEvent;

private void OnPusherAuthenticationEvent(TObject *Sender, TsgcWSPusherRequestAuthentication
  *AuthRequest, TsgcWSPusherResponseAuthentication *AuthResponse)
{
  // if the authentication request is successful return the signature
  if (CustomAuthentication(AuthRequest->Channel, AuthRequest->SocketID))
  {
    AuthResponse->Signature = GetCustomAuthenticationSignature();
  }
}
```

The format of the signature is:
Private channels: key:HMAC256(SocketID, ChannelName)
Presence channels: key: HMAC256(SocketID, ChannelName, Data)
The **TsgcWSPusherResponseAuthentication** object provides the following properties:

Property	Description
Secret	The Pusher secret key used to compute the HMAC signature. Pre-filled with Pusher.Secret if configured.
Signature	The computed authentication signature. If left empty, the component will calculate it automatically using the Secret.
SharedSecret	The shared secret key for private-encrypted channels. Required when subscribing to pscPrivateEncryptedChannel or pscPrivateEncryptedCacheChannel. Used for end-to-end encryption of message data.

API Bitmex

Bitmex

Bitmex is a cryptocurrency exchange and derivative trading platform.

The following APIs are supported:

1. **WebSocket streams:** allows you to subscribe to some methods and get data in real-time. Events are pushed to clients by server to subscribers. Uses WebSocket as protocol.
2. **REST API:** clients can request to server market and account data. Requires an API Key and Secret to authenticate and uses HTTPs as protocol.

Properties

Bitmex API has 2 types of methods: public and private. Public methods can be accessed without authentication, for example: get ticker prices. Some are private and related to user data; those methods require the use of Bitmex API keys.

- **ApiKey:** you can request a new api key in your Bitmex account, just copy the value to this property.
- **ApiSecret:** it's the secret of the API, keep safe.
- **TestNet:** if enabled it will connect to Bitmex Demo Account (by default false).
- **HTTPLogOptions:** stores in a text file a log of HTTP requests
  - **Enabled:** if enabled, will store all HTTP requests of WebSocket API.
  - **FileName:** full path of filename where logs will be stored
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Most common uses

- **WebSockets API**
  - **How to Connect to WebSocket API**
  - **How to Subscribe to a WebSocket Channel**
- **REST API**
  - **How to Place a Bitmex Order**

WebSocket API

Subscribe / Unsubscribe

BitMEX allows subscribing to real-time data. This access is not rate-limited once connected and is the best way to get the most up-to-date data to your programs. In some topics, you can pass a Symbol to filter events by symbol, example: trades, quotes...

The following subscription topics are available without authentication:

- **btmAnnouncement**: Site Announcements
- **btmChat**: Trollbox chat
- **btmConnected**: Statistics of connected users/bots
- **btmFunding**: Updates of swap funding rates. Sent every funding interval (usually 8hrs)
- **btmInstrument**: Instrument updates including turnover and bid/ask
- **btmInsurance**: Daily Insurance Fund updates
- **btmLiquidation**: Liquidation orders as they're entered into the book
- **btmOrderBookL2_25**: Top 25 levels of level 2 order book

- **btmOrderBookL2**: Full level 2 order book
- **btmOrderBook10**: Top 10 levels using traditional full book push
- **btmPublicNotifications**: System-wide notifications (used for short-lived messages)
- **btmQuote**: Top level of the book
- **btmQuoteBin1m**: 1-minute quote bins
- **btmQuoteBin5m**: 5-minute quote bins
- **btmQuoteBin1h**: 1-hour quote bins
- **btmQuoteBin1d**: 1-day quote bins
- **btmSettlement**: Settlements
- **btmTrade**: Live trades
- **btmTradeBin1m**: 1-minute trade bins
- **btmTradeBin5m**: 5-minute trade bins
- **btmTradeBin1h**:1-hour trade bins
- **btmTradeBin1d**: 1-day trade bins

The following subjects require authentication:

- **btmAffiliate**: Affiliate status, such as total referred users & payout %
- **btmExecution**: Individual executions; can be multiple per order
- **btmOrder**: Live updates on your orders
- **btmMargin**: Updates on your current account balance and margin requirements
- **btmPosition**: Updates on your positions
- **btmPrivateNotifications**: Individual notifications - currently not used
- **btmTransact**: Deposit/Withdrawal updates
- **btmWallet**: Bitcoin address balance data, including total deposits & withdrawals

Example of messages received:


```
{
      "table":"orderBookL2_25",
      "keys":["symbol","id","side"],
      "types":{"id":"long","price":"float","side":"symbol","size":"long","symbol":"symbol"}
      "foreignKeys":{"side":"side","symbol":"instrument"},
      "attributes":{"id":"sorted","symbol":"grouped"},
      "action":"partial",
      "data":[
        {"symbol":"XBTUSD","id":17999992000,"side":"Sell","size":100,"price":80},
        {"symbol":"XBTUSD","id":17999993000,"side":"Sell","size":20,"price":70},
        {"symbol":"XBTUSD","id":17999994000,"side":"Sell","size":10,"price":60},
        {"symbol":"XBTUSD","id":17999995000,"side":"Buy","size":10,"price":50},
        {"symbol":"XBTUSD","id":17999996000,"side":"Buy","size":20,"price":40},
        {"symbol":"XBTUSD","id":17999997000,"side":"Buy","size":100,"price":30}
      ]
}

{
      "table":"orderBookL2_25",
      "action":"update",
      "data":[
        {"symbol":"XBTUSD","id":17999995000,"side":"Buy","size":5}
      ]
    }
{
      "table":"orderBookL2_25",
      "action":"delete",
      "data":[
        {"symbol":"XBTUSD","id":17999995000,"side":"Buy"}
      ]
}
{
      "table":"orderBookL2_25",
      "action":"insert",
      "data":[
        {"symbol":"XBTUSD","id":17999995500,"side":"Buy","size":10,"price":45},
      ]
}
```


Authentication

If you wish to subscribe to user-locked streams, you must authenticate first. Note that invalid authentication will close the connection.

BitMEX API usage requires an API Key.

Permanent API Keys can be locked to IP address ranges and revoked at will without compromising your main credentials. They also do not require renewal.

To use API Key auth, you must generate an API Key in your account.

Call method **Authenticate** before subscribing to any Authenticated Topic.

CancelAllAfter (Dead Man's Switch)

The **CancelAllAfter** method implements the Dead Man's Switch feature. When called with a timeout value (in milliseconds), it instructs the server to cancel all open orders if no subsequent CancelAllAfter call is received within the timeout period. This is useful to ensure orders are canceled in case of network disconnection.

REST API

Method	Description
GetExecutions	This returns all raw transactions, which includes order opening and cancelation, and order status changes.
GetExecutionsTradeHistory	This returns more focused Transactions.
GetInstruments	This returns all instruments and indices, including those that have settled or are unlisted. Use this endpoint if you want to query for individual instruments or use a complex filter.
GetOrders	To get open orders only
PlaceOrder	Place a raw order using TsgcHTTPBitmexOrder object.
PlaceMarketOrder	Place a new MARKET order.
PlaceLimitOrder	Place a new LIMIT order.
PlaceStopOrder	Place a new STOP order.
PlaceStopLimitOrder	Place a new STOPLIMIT order.
AmendOrder	Modify an existing order.
CancelOrder	Cancels an active Order.
CancelAllOrders	Cancel All Active Orders.
CancelAllOrdersAfter	Cancel All Orders after some time.
ClosePosition	Close an open position.
GetOrderBook	Get Current OrderBook in vertical format
GetPosition	Get your positions.
SetPositionIsolate	Enable isolated margin or cross-margin per position.
SetPositionLeverage	Choose leverage per position.
SetPositionRiskLimit	Update your risk limit.
SetPositionTransferMargin	Transfer equity in or out of a position.
GetQuotes	Get Quotes
GetTrades	Get Trades
GetFunding	Get funding data.
GetInsurance	Get insurance fund data.
GetTradeBucketed	Get bucketed trade data (OHLCV) with configurable bin sizes.

GetQuoteBucketed	Get bucketed quote data with configurable bin sizes.
GetSettlement	Get settlement data.
GetLiquidation	Get liquidation orders.
GetInstrumentIndices	Get instrument indices.
GetInstrumentCompositeIndex	Get composite index data for instruments.
GetStats	Get exchange-wide statistics.
GetStatsHistory	Get historical exchange statistics.
GetStatsHistoryUSD	Get historical USD exchange statistics.
GetUserMargin	Get your account margin data.
GetUserWallet	Get your wallet information.
GetUserWalletHistory	Get your wallet transaction history.
GetUserWalletSummary	Get a summary of your wallet.

Bitmex | Connect WebSocket API

In order to connect to Bitmex WebSocket API, just create a new Bitmex API client and attach to TsgcWebSocket-Client.
See below an example:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bitmex oBitmex = new TsgcWSAPI_Bitmex();
oBitmex->Client = oClient;
oClient->Active = true;
```

Bitmex | Subscribe WebSocket Channel

Bitmex offers a variety of channels where you can subscribe to get real-time updates of market data, orders... Find below a sample of how subscribe to a Trade Channel:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bitmex oBitmex = new TsgcWSAPI_Bitmex();
oBitmex->Client = oClient;
oBitmex->Subscribe(btmTrade, "XBTUSD");
void OnBitmexMessage(Sender: TObject; const aTopic: TwsBitmexTopics; const aMessage: string)
{
// here you will receive the trade updates
}
```

Bitmex | How to Place Orders

The Bitmex REST API offer public and private endpoints. The Private endpoints require that messages are signed to increase the security of transactions.
First you must login to your Bitmex account and create a new API, you will get the following values:

- ApiKey
- ApiSecret

These fields must be configured in the Bitmex property of the Bitmex API client component. Once configured, you can start to do private requests to the Bitmex REST API.

Order Types

All orders require a symbol. All other fields are optional except when otherwise specified.
These are the valid ordTypes:

- **Limit:** The default order type. Specify an orderQty and price.
- **Market:** A traditional Market order. A Market order will execute until filled or your bankruptcy price is reached, at which point it will cancel.
- **Stop:** A Stop Market order. Specify an orderQty and stopPx. When the stopPx is reached, the order will be entered into the book.
    - On sell orders, the order will trigger if the triggering price is lower than the stopPx. On buys, higher.
    - Note: Stop orders do not consume margin until triggered. Be sure that the required margin is available in your account so that it may trigger fully.
    - Close Stops don't require an orderQty. See Execution Instructions below.
- **StopLimit:** Like a Stop Market, but enters a Limit order instead of a Market order. Specify an orderQty, stopPx, and price.
- **MarketIfTouched:** Similar to a Stop, but triggers are done in the opposite direction. Useful for Take Profit orders.
- **LimitIfTouched:** As above; use for Take Profit Limit orders.
- **Pegged:** Pegged orders allow users to submit a limit price relative to the current market price. Specify a pegPriceType, and pegOffsetValue.
    - Pegged orders must have an execInst of Fixed. This means the limit price is set at the time the order is accepted and does not change as the reference price changes.
    - PrimaryPeg: Price is set relative to near touch price.
    - MarketPeg: Price is set relative to far touch price.
    - A pegPriceType submitted with no ordType is treated as a Pegged order.

Execution Instructions

The following execInsts are supported. If using multiple, separate with a comma (e.g. LastPrice,Close).

- **ParticipateDoNotInitiate:** Also known as a Post-Only order. If this order would have executed on placement, it will cancel instead. This is intended to protect you from the far touch moving towards you while the order is in transit. It is not intended for speculating on the far touch moving away after submission - we consider such behaviour abusive and monitor for it.
- **MarkPrice, LastPrice, IndexPrice:** Used by stop and if-touched orders to determine the triggering price. Use only one. By default, MarkPrice is used. Also used for Pegged orders to define the value of LastPeg.
- **ReduceOnly:** A ReduceOnly order can only reduce your position, not increase it. If you have a ReduceOnly limit order that rests in the order book while the position is reduced by other orders, then its order quantity will be amended down or canceled. If there are multiple ReduceOnly orders the least aggressive will be amended first.
- **Close:** Close implies ReduceOnly. A Close order will cancel other active limit orders with the same side and symbol if the open quantity exceeds the current position. This is useful for stops: by canceling these orders, a Close Stop is ensured to have the margin required to execute, and can only execute up to the full size of your position. If orderQty is not specified, a Close order has an orderQty equal to your current position's size.

    - Note that a Close order without an orderQty requires a side, so that BitMEX knows if it should trigger above or below the stopPx.
- **LastWithinMark:** Used by stop orders with LastPrice to allow stop triggers only when:
    - For Sell Stop Market / Stop Limit Order
        - Last Price <= Stop Price
        - Last Price >= Mark Price × (1 - 5%)
    - For Buy Stop Market / Stop Limit Order:
        - Last Price >= Stop Price
        - Last Price <= Mark Price × (1 + 5%)
- **Fixed:** Pegged orders must have an execInst of Fixed. This means the limit price is set at the time the order is accepted and does not change as the reference price changes.

Pegged Orders

Pegged orders allow users to submit a limit price relative to the current market price. The limit price is set once when the order is submitted and does not change with the reference price. This order type is not intended for speculating on the far touch moving away after submission - we consider such behaviour abusive and monitor for it.

Pegged orders have an ordType of Pegged, and an execInst of Fixed.

A pegPriceType and pegOffsetValue must also be submitted:

- PrimaryPeg - price is set relative to the near touch price
- MarketPeg - price is set relative to the far touch price

Trailing Stop Pegged Orders

Use pegPriceType of TrailingStopPeg to create Trailing Stops.

The price is set at submission and updates once per second if the underlying price (last/mark/index) has moved by more than 0.1%. stopPx then moves as the market moves away from the peg, and freezes as the market moves toward it.

Use pegOffsetValue to set the stopPx of your order. The peg is set to the triggering price specified in the execInst (default MarkPrice). Use a negative offset for stop-sell and buy-if-touched orders.

Requires ordType: Stop, StopLimit, MarketIfTouched, LimitIfTouched.

Trailing Stops

You may use pegPriceType of 'TrailingStopPeg' to create Trailing Stops. The pegged stopPx will move as the market moves away from the peg, and freeze as the market moves toward it.

To use, combine with pegOffsetValue to set the stopPx of your order. The peg is set to the triggering price specified in the execInst (default 'MarkPrice'). Use a negative offset for stop-sell and buy-if-touched orders.

Requires ordType: 'Stop', 'StopLimit', 'MarketIfTouched', 'LimitIfTouched'.

Tracking Your Orders

If you want to keep track of order IDs yourself, set a unique clOrdID per order. This clOrdID will come back as a property on the order and any related executions (including on the WebSocket), and can be used to get or cancel the order. Max length is 36 characters.

Examples:

```
// buy market order
BITMEX->REST_API->PlaceMarketOrder(bmosBuy, "XBTUSD", 100);
// sell limit order at 45000
BITMEX->REST_API->PlaceLimitOrder(bmosSell, "XBTUSD", 100, 45000.00);
// stop order at 48000
BITMEX->REST_API->PlaceStopOrder(bmosSell, "XBTUSD", 100, 48000.00);
```

API Bitfinex

Bitfinex

Bitfinex is one of the world's largest and most advanced cryptocurrency trading platform. Users can exchange Bitcoin, Ethereum, Ripple, EOS, Bitcoin Cash, Iota, NEO, Litecoin, Ethereum Classic...

Bitfinex WebSocket API version is 2.0

Each message sent and received via the Bitfinex's WebSocket channel is encoded in JSON format

A symbol can be a trading pair or a margin currency:

- Trading pairs symbols are formed prepending a "t" before the pair (i.e tBTCUSD, tETHUSD).
- Margin currencies symbols are formed prepending an "f" before the currency (i.e fUSD, fBTC, ...)

After a successful connection, **OnBitfinexConnect** event is raised and you get Bitfinex API Version number as a parameter.

You can call **Ping** method to test connection to the server. The server replies with a pong message, which raises **OnBitfinexPong** event with the Cid of the request and the server Timestamp (Ts, in milliseconds).

If the server sends any information, this can be handled using **OnBitfinexInfoMsg** event, where a Code and a Message are parameters with information about the message sent by the server. Example codes:

20051 : Stop/Restart WebSocket Server (please reconnect)
20060 : Entering in Maintenance mode. Please pause any activity and resume after receiving the info message 20061 (it should take 120 seconds at most).
20061 : Maintenance ended. You can resume normal activity. It is advised to unsubscribe/subscribe again all channels.

In case of error, **OnBitfinexError** will be raised, and information about error provided. Example error codes:

10000 : Unknown event
10001 : Unknown pair

In order to change the configuration, call **Configuration** method and pass as a parameter one of the following flags. The server confirms the new configuration raising **OnBitfinexConf** event with the Status and the Flags applied.

CS_DEC_S = 8; // Enable all decimal as strings.
CS_TIME_S = 32; // Enable all times as date strings.
CS_TIMESTAMP = 32768; // Adds a timestamp in milliseconds to each received event.
CS_SEQ_ALL = 65536; // Enable sequencing BETA FEATURE
CHECKSUM = 131072; // Enable checksum for every book iteration. Checks the top 25 entries for each side of the book. The checksum is a signed int.
CS_BULK_UPDATES = 536870912; // Enables receiving multiple book updates in a single message as an array of arrays.

Properties

- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Subscribe Public Channels

There are channels which are public and there is no need to authenticate against the server. All messages are raised **OnBitfinexUpdate** event.

SubscribeTicker

The ticker is a high level overview of the state of the market. It shows you the current best bid and ask, as well as the last trade price. It also includes information such as daily volume and how much the price has moved over the last day.

```
// Trading pairs
[
  CHANNEL_ID,
  [
    BID,
    BID_SIZE,
    ASK,
    ASK_SIZE,
    DAILY_CHANGE,
    DAILY_CHANGE_PERC,
    LAST_PRICE,
    VOLUME,
    HIGH,
    LOW
  ]
]
// Funding pairs
[
  CHANNEL_ID,
  [
    FRR,
    BID,
    BID_PERIOD,
    BID_SIZE,
    ASK,
    ASK_PERIOD,
    ASK_SIZE,
    DAILY_CHANGE,
    DAILY_CHANGE_PERC,
    LAST_PRICE,
    VOLUME,
    HIGH,
    LOW
  ]
]
```

SubscribeTrades

This channel sends a trade message whenever a trade occurs at Bitfinex. It includes all the pertinent details of the trade, such as price, size and time.

```
// on trading pairs (ex. tBTCUSD)
[
  CHANNEL_ID,
  [
    [
      ID,
      MTS,
      AMOUNT,
      PRICE
    ],
    ...
  ]
]
// on funding currencies (ex. fUSD)
[
  CHANNEL_ID,
  [
    [
      ID,
      MTS,
      AMOUNT,
      RATE,
      PERIOD
```

```
    ],
    ...
  ]
]
```

SubscribeOrderBook

The Order Books channel allows you to keep track of the state of the Bitfinex order book. It is provided on a price aggregated basis, with customizable precision. After receiving the response, you will receive a snapshot of the book, followed by updates upon any changes to the book.

```
// on trading pairs (ex. tBTCUSD)
[
  CHANNEL_ID,
  [
    [
      PRICE,
      COUNT,
      AMOUNT
    ],
    ...
  ]
]

// on funding currencies (ex. fUSD)
[
  CHANNEL_ID,
  [
    [
      RATE,
      PERIOD,
      COUNT,
      AMOUNT
    ],
    ...
  ]
]
```

SubscribeRawOrderBook

These are the most granular books.

```
// on trading pairs (ex. tBTCUSD)
[
  CHANNEL_ID,
  [
    [
      ORDER_ID,
      PRICE,
      AMOUNT
    ],
    ...
  ]
]

// on funding currencies (ex. fUSD)
[
  CHANNEL_ID,
  [
    [
      OFFER_ID,
      PERIOD,
      RATE,
      AMOUNT
    ],
    ...
  ]
]
```

SubscribeCandles

Provides a way to access charting candle info. Time Frames:

1m: one minute
5m : five minutes
15m : 15 minutes
30m : 30 minutes
1h : one hour
3h : 3 hours
6h : 6 hours
12h : 12 hours
1D : one day
7D : one week
14D : two weeks
1M : one month

```
[
  CHANNEL_ID,
  [
    [
      MTS,
      OPEN,
      CLOSE,
      HIGH,
      LOW,
      VOLUME
    ],
    ...
  ]
]
```

Subscribe Authenticated Channels

This channel allows you to keep up to date with the status of your account. You can receive updates on your positions, your balances, your orders and your trades.

Use **Authenticate** method in order to Authenticate against the server and set required parameters.

Once authenticated, you will receive updates of: Orders, positions, trades, funding offers, funding credits, funding loans, wallets, balance info, margin info, funding info, funding trades...

You can request **UnAuthenticate** method if you want to log off from the server.

API Kucoin

Kucoin

Kucoin is an international multi-language cryptocurrency exchange. It offers some APIs to access Kucoin data. The following APIs are supported:

1. **WebSocket streams:** allows you to subscribe to some methods and get data in real-time. Events are pushed to clients by server to subscribers. Uses WebSocket as protocol.
2. **REST API:** clients can request to server market and account data. Requires an API Key, Secret and Passphrase to authenticate and uses HTTPs as protocol.

Properties

Kucoin API has 2 types of methods: public and private. Public methods can be accessed without authentication, example: get ticker prices. Private methods related to user data require the use of Kucoin API keys.

- **ApiKey:** you can request a new api key in your kucoin account, just copy the value to this property.
- **ApiSecret:** API secret is only required for REST_API, websocket api only requires ApiKey for some methods.
- **Passphrase:** string required to connect to Kucoin Servers.
- **Sandbox:** if enabled it will connect to Kucoin Demo Account (by default false).
    - **HTTPLogOptions:** stores in a text file a log of HTTP requests
        - **Enabled:** if enabled, will store all HTTP requests of WebSocket API.
        - **FileName:** full path of filename where logs will be stored
        - **REST:** stores in a text file a log of REST API requests
            - **Enabled:** if enabled, will store all HTTP Requests of REST API.
            - **FileName:** full path of filename where logs will be stored.
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Most common uses

- **WebSockets API**
    - **How to Connect to WebSocket API**
    - **How to Subscribe to a WebSocket Channel**
- **REST API**
    - **How to Get Market Data**
    - **How to Use Private REST API**
    - **How to Trade Spot**
    - **Private Requests Time**

WebSocket Feed

To subscribe channel messages from a certain server, the client side should send subscription message to the server.
If the subscription succeeds, the system will send ack messages to you, when the response is set as true.

```
{
   "id":"1545910660739",
   "type":"ack"
}
```

While there are topic messages generated, the system will send the corresponding messages to the client side.

The following Subscription / Unsubscription methods are supported.

Public Channels

Method	Parameters	Description
SubscribeSymbolTicker	Symbol	Subscribe to this topic to get the push of BBO changes. If there is no change within one second, it will not be pushed. It will be pushed per 100ms with the newest BBO. If there was no change compared with last data, it will not be pushed.
SubscribeAllSymbolsTicker		Subscribe to this topic to get the push of all market symbols BBO change.
SubscribeSymbolSnapshot	Symbol	Subscribe to get snapshot data for a single symbol. The snapshot data is pushed at 2 seconds intervals.
SubscribeMarketSnapshot	Market	Subscribe this topic to get the snapshot data for the entire market. The snapshot data is pushed at 2 seconds intervals.
SubscribeLevel2MarketData	Symbol	Subscribe to this topic to get Level2 order book data. When the websocket subscription is successful, the system would send the increment change data pushed by the websocket to you.
SubscribeLevel2_5BestAskBid	Symbol	The system will return the 5 best ask/bid orders data, which is the snapshot data of every 100 milliseconds (in other words, the 5 best ask/bid orders data returned every 100 milliseconds in real-time).
SubscribeLevel2_50BestAskBid	Symbol	The system will return the 50 best ask/bid orders data, which is the snapshot data of every 100 milliseconds (in other words, the 50 best ask/bid orders data returned every 100 milliseconds in real-time).
SubscribeKlines	Symbol	Subscribe to this topic to get K-Line data.
SubscribeMatchExecutionData	Symbol	Subscribe to this topic to get the matching event data flow of Level 3. For each order traded, the system would send you the match messages in the following format.
SubscribeIndexPrice	Symbol	Subscribe to this topic to get the index price for the margin trading.
SubscribeMarkPrice	Symbol	Subscribe to this topic to get the mark price for margin trading.
SubscribeOrderBookChanged	Symbol	Subscribe to this topic to get the order book changes on margin trade.
SubscribeLevel1	Symbol	Subscribe to Level 1 best bid/ask data for a symbol.

If ACK parameter is sent to true, after a successful subscription / unsubscription, client receives a message about it.

Private Channels

Requires a valid ApiKey obtained from your Kucoin account. The ApiKey, ApiSecret and Passphrase must be set in the Kucoin property of the client API component.

The following data is pushed to client every time there is a change. There is no need to subscribe to any method, this is done automatically if you set a valid ApiKey.

Method	Description
SubscribeTradeOrders	This topic will push all change events of your orders.

SubscribeAccountBalance	You will receive this message when an account balance changes. The message contains the details of the change.
SubscribePositionStatus	The system will push the change event when the position status changes.
SubscribeMarginTradeOrders	The system will push this message to the lenders when the order enters the order book.
SubscribeStopOrder	When a stop order is received by the system, you will receive a message with "open" type. It means that this order entered the system and waited to be triggered.
SubscribeTradeOrdersV2	Subscribe to trade orders V2 channel for enhanced order update notifications.
SubscribeCrossMarginPosition	Subscribe to cross margin position updates. The system will push the change event when the cross margin position changes.
SubscribeIsolatedMarginPosition	Subscribe to isolated margin position updates. The system will push the change event when the isolated margin position changes.

REST API

All endpoints return either a JSON object or array.

Public API EndPoints

These endpoints can be accessed without any authorization.

General EndPoints

Method	Parameters	Description
GetServiceStatus		Test connectivity to the Rest API and get the Service Status
GetServerTime		Test connectivity to the Rest API and get the current server time.

Market Data EndPoints

Method	Parameters	Description
GetSymbolList	Market	Request via this endpoint to get a list of available currency pairs for trading. If you want to get the market information of the trading symbol
GetTicker	Symbol	Request via this endpoint to get Level 1 Market Data. The returned value includes the best bid price and size, the best ask price and size as well as the last traded price and the last traded size.
GetAllTickers		Request market tickers for all the trading pairs in the market (including 24h volume).
Get24hrStats	Symbol	Request via this endpoint to get the statistics of the specified ticker in the last 24 hours.
GetMarketList		Request via this endpoint to get the transaction currency for the entire trading market.

GetPartOrder-Book20	Symbol	Request via this endpoint to get a list of open orders for a symbol. Level-2 order book includes all bids and asks (aggregated by price), this level returns only one size for each active price (as if there was only a single order for that price). The system will return you 20 pieces of data (ask and bid data) on the order book.
GetPartOrder-Book100	Symbol	Request via this endpoint to get a list of open orders for a symbol. Level-2 order book includes all bids and asks (aggregated by price), this level returns only one size for each active price (as if there was only a single order for that price). The system will return you 100 pieces of data (ask and bid data) on the order book.
GetFullOrder-Book	Symbol	Request via this endpoint to get the order book of the specified symbol. Level 2 order book includes all bids and asks (aggregated by price). This level returns only one aggregated size for each price (as if there was only one single order for that price). This API will return data with full depth.
GetKLines	Symbol	Request via this endpoint to get the kline of the specified symbol. Data are returned in grouped buckets based on requested type.
GetCurrencies		Request via this endpoint to get the currency list.
GetCurrency-Detail	Currency	Request via this endpoint to get the currency details of a specified currency
GetFiatPrice		Request via this endpoint to get the currency details of a specified currency
GetPartOrder-Book1	Symbol	Request via this endpoint to get the Level 1 best bid/ask for a symbol.

Private API EndPoints

Requires an APIKey and APISecret to get authorized by server.

User EndPoints

Method	Parameters	Description
GetAllSubAc-counts		You can get the user info of all sub-users via this interface.
GetListAc-counts		Get a list of accounts.
GetAccount	AccountId	Information for a single account. Use this endpoint when you know the accountId.
GetAccountBal-anceSubAc-count	SubUserId	This endpoint returns the account info of a sub-user specified by the subUserId.
InnerTransfer		This API endpoint can be used to transfer funds between accounts internally. Users can transfer funds between their main account, trading account, cross margin account, and isolated margin account free of charge. Transfer of funds from the main account, cross margin account, and trading account to the futures account is supported, but transfer of

		funds from futures accounts to other accounts is not supported.
GetDepositAddresses	Currency	Get deposit addresses for a currency.
CreateDepositAddress	Currency	Create a new deposit address for a currency.
GetDepositList		Get deposit history.
GetAccountLedgers		Get account ledger entries.
GetTradeFees	Symbols	Get trade fees for the specified symbols.

Withdraw EndPoints

Method	Parameters	Description
GetWithdrawalsList		Get a list of the Withdrawals.
GetHistoricalWithdrawalsList		List of KuCoin V1 historical withdrawals.
GetWithdrawalsQuotas	Currency	Get Withdrawals Quotas
ApplyWithdraw	Currency, Address, Amount	Create a Withdraw
CancelWithdraw	WithdrawalId	Only withdrawals requests of PROCESSING status could be canceled.

Trade Endpoints

Method	Parameters	Description
PlaceOrder		You can place two types of orders: limit and market. Orders can only be placed if your account has sufficient funds. Once an order is placed, your account funds will be put on hold for the duration of the order. How much and which funds are put on hold depends on the order type and parameters specified
PlaceMarketOrder		Places a Market Order.
PlaceLimitOrder		Places a Limit Order.
PlaceMarginOrder		Places a Margin Order.
CancelOrder		Cancels an Order by Order Id.
CancelOrderByClientOid		Cancels an Order by Client Order Id.
CancelAllOrders		Cancel all open orders.
ListOrders		Request via this endpoint to get your current order list. Items are paginated and sorted to show the latest first

GetRecentOrders		Request via this endpoint to get 1000 orders in the last 24 hours.
GetOrder		Request via this endpoint to get a single order info by order ID.
GetOrderByClientOid		Request via this endpoint to get a single order info by Client order ID.
ListFills		Request via this endpoint to get the recent fills.
GetRecentFills		Request via this endpoint to get a list of 1000 fills in the last 24 hours.
PlaceStopOrder		Places a Stop Order.
PlaceStopMarketOrder		Places a Stop Market Order.
PlaceStopLimitOrder		Places a Stop Limit Order.
CancelStopOrder		Cancels a Open Stop Order by Order Id
CancelStopOrderByClientOid		Cancels a Open Stop Order by Client Order Id
CancelAllStopOrders		Cancel All Stop Orders
GetStopOrder		Request via this interface to get a stop order information via the order ID.
GetStopOrderByClientOid		Request via this interface to get a stop order information via the Client order ID.
ListStopOrders		Request via this endpoint to get your current untriggered stop order list. Items are paginated and sorted to show the latest first.
PlaceHFOrder		Place a high-frequency order.
CancelHFOrder		Cancel a high-frequency order by order ID.
CancelHFOrderByClientOid		Cancel a high-frequency order by client order ID.
CancelAllHFOrders		Cancel all high-frequency orders.
GetHFActiveOrders		Get active high-frequency orders.
GetHFDoneOrders		Get completed high-frequency orders.
GetHFOrder		Get a specific high-frequency order by order ID.

Events

Kucoin Messages are received in TsgcWebSocketClient component, you can use the following events:

OnConnect
After a successful connection to Kucoin server.
OnDisconnect
After a disconnection from Kucoin server
OnMessage
Messages sent by server to client are handled in this event.

OnError
If there is any error in protocol, this event will be called.
OnException
If there is an unhandled exception, this event will be called.

Additionally, there is a specific event in Kucoin API Component, called **OnKucoinHTTPException**, which is raised every time there is an error calling an HTTP Request (REST API or WebSocket Feeds).

Kucoin | Connect WebSocket API

In order to connect to Kucoin WebSocket API, just create a new Kucoin API client and attach to TsgcWebSocketClient.
See below an example:

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin();
oKucoin->Client = oClient;
oClient->Active = true;
```

Kucoin | Subscribe WebSocket Channel

Kucoin offers a variety of channels where you can subscribe to get real-time updates of market data, orders... Find below a sample of how to subscribe to a Ticker:

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin();
oKucoin->Client = oClient;
oKucoin->SubscribeSymbolTicker("BTC-USDT");
void OnMessage(TsgcWSConnection *Connection, const string aText)
{
// here you will receive the ticker updates
}
```

Kucoin | Get Market Data

Kucoin offers public Market Data through REST Endpoints, when you call one of these endpoints, you will get a snapshot of the market data requested.
The Market Data Endpoints don't require authentication, so are freely available to all users.

Example: to get the snapshot of the ticker BTC-USDT, do the following call

```
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin(this);
ShowMessage(oKucoin->REST_API->GetTicker("

BTC-USDT

"));
```

Kucoin | Private REST API

The Kucoin REST API offer public and private endpoints. The Private endpoints require that messages are signed to increase the security of transactions.
First you must login to your Kucoin account and create a new API, you will get the following values:

- ApiKey
- ApiSecret
- Passphrase

These fields must be configured in the Kucoin property of the Kucoin API client component.
Once configured, you can start to do private requests to the Kucoin Pro REST API

*Private Requests, require that your local machine has the local time synchronized, if not, the requests will be rejected by Kucoin server. Check the following article about this, Kucoin Private Requests Time.

```
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin(this);
oKucoin->Kucoin->ApiKey = "<your api key>";
oKucoin->Kucoin->ApiSecret = "<your api secret>";
oKucoin->Kucoin->Passphrase = "<your passphrase>";
ShowMessage(oKucoin->REST_API->GetListAccounts());
```

Kucoin | Trade Spot

Kucoin allows you to trade spot using its REST API.

Configuration

First you must create an **API Key** in your Kucoin account and add privileges to trading with Spot.
Once this is done, you can start spot trading.
First, **set your ApiKey, ApiSecret and Passphrase** in the Kucoin Client Component, this will be used to sign the requests sent to Kucoin server.

Place an Order

To place a new order, just call to method **REST_API.PlaceOrder** of Kucoin Client Component.

Depending on the type of the order (market, limit...) the API requires more or less fields.

Parameters

Param	type	Description
clientOid	String	Unique order id created by users to identify their orders, e.g. UUID.
side	String	**buy** or **sell**
symbol	String	a valid trading symbol code. e.g. ETH-BTC
type	String	*[Optional]* **limit** or **market** (default is **limit**)
remark	String	*[Optional]* remark for the order, length cannot exceed 100 utf8 characters
stp	String	*[Optional]* self trade prevention , **CN**, **CO**, **CB** or **DC**
trade-Type	String	*[Optional]* The type of trading : **TRADE**（Spot Trade）, **MARGIN_TRADE** (Margin Trade). Default is **TRADE**. Note: To improve the system performance and to accelerate order placing and processing, KuCoin has added a new interface for order placing of margin. For traders still using the current interface, please move to the new one as soon as possible. The current one will no longer accept margin orders by May 1st, 2021 (UTC). At the time, KuCoin will notify users via the announcement, please pay attention to it.

LIMIT ORDER PARAMETERS

Param	type	Description
price	String	price per base currency
size	String	amount of base currency to buy or sell
timeInForce	String	*[Optional]* **GTC**, **GTT**, **IOC**, or **FOK** (default is **GTC**), read Time In Force.
cancelAfter	long	*[Optional]* cancel after **n** seconds, requires **timeInForce** to be **GTT**
postOnly	boolean	*[Optional]* Post only flag, invalid when **timeInForce** is **IOC** or **FOK**
hidden	boolean	*[Optional]* Order will not be displayed in the order book

Param	type	Description
iceberg	boolean	*[Optional]* Only a portion of the order is displayed in the order book
visibleSize	String	*[Optional]* The maximum visible size of an iceberg order

MARKET ORDER PARAMETERS

Param	type	Description
size	String	*[Optional]* Desired amount in base currency
funds	String	*[Optional]* The desired amount of quote currency to use

When you send an order, there are 2 possibilities:

1. **Successful:** the function PlaceOrder returns the message sent by Kucoin server.
2. **Error:** the exception is returned in the event OnKucoinHTTPException.

Place Market Order 1 BTC-USDT

```
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin(this);
oKucoin->Kucoin->ApiKey = "<api key>";
oKucoin->Kucoin->ApiSecret = "<api secret>";
oKucoin->Kucoin->Passphrase = "<passphrase>";
ShowMessage(oKucoin->REST_API->

PlaceMarketOrder

(

kosBuy, "BTC-USDT", 1

));
```

Place Limit Order 1 BTC-USDT at 40000

```
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin(this);
oKucoin->Kucoin->ApiKey = "<api key>";
oKucoin->Kucoin->ApiSecret = "<api secret>";
oKucoin->Kucoin->Passphrase = "<passphrase>";
ShowMessage(oKucoin->REST_API->

PlaceLimitOrder

(

kosBuy, "BTC-USDT", 1, 40000

));
```

Kucoin | Private Requests Time

When you do a private request to Kucoin, the message is signed to increase the security of requests. The message takes the local time and sends inside the signed message, if the local time has a difference greater than 5 seconds with Kucoin servers, the request will be rejected. So, it's important to verify that your local time is synchronized, you can do this using the synchronization time method for your OS.

You can check the Kucoin server time, calling method **GetServerTime**, which will return the time of the Kucoin server

API Kucoin Futures

Kucoin Futures

Kucoin is an international multi-language cryptocurrency exchange. It offers some APIs to access Kucoin data. The following APIs are supported:

1. **WebSocket streams:** allows you to subscribe to some methods and get data in real-time. Events are pushed to clients by server to subscribers. Uses WebSocket as protocol.
2. **REST API:** clients can request to server market and account data. Requires an API Key, Secret and Passphrase to authenticate and uses HTTPs as protocol.

Properties

Kucoin API has 2 types of methods: public and private. Public methods can be accessed without authentication, example: get ticker prices. Private and related to user data methods require the use of Kucoin API keys.

- **ApiKey:** you can request a new api key in your kucoin account, just copy the value to this property.
- **ApiSecret:** API secret is only required for REST_API, websocket api only requires ApiKey for some methods.
- **Passphrase:** string required to connect to Kucoin Servers.
- **Sandbox:** if enabled it will connect to Kucoin Demo Account (by default false).
    - **HTTPLogOptions:** stores in a text file a log of HTTP requests
        - **Enabled:** if enabled, will store all HTTP requests of WebSocket API.
        - **FileName:** full path of filename where logs will be stored
        - **REST:** stores in a text file a log of REST API requests
            - **Enabled:** if enabled, will store all HTTP Requests of REST API.
            - **FileName:** full path of filename where logs will be stored.
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Most common uses

- **WebSockets API**
    - **How to Connect to WebSocket API**
    - **How to Subscribe to a WebSocket Channel**
- **REST API**
    - **How to Get Market Data**
    - **How to Use Private REST API**
    - **How to Trade Futures**
    - **Private Requests Time**

WebSocket Feed

To subscribe channel messages from a certain server, the client side should send subscription message to the server.
If the subscription succeeds, the system will send ack messages to you, when the response is set as true.

```
{
   "id":"1545910660739",
   "type":"ack"
}
```

While there are topic messages generated, the system will send the corresponding messages to the client side.

The following Subscription / Unsubscription methods are supported.

Public Channels

Method	Parameters	Description
SubscribeSymbolTickerV2	Symbol	Subscribe this topic to get the realtime push of BBO changes. After subscription, when there are changes in the order book, the system will push the real-time ticker symbol information to you. It is recommended to use the new topic for timely information.
SubscribeSymbolTicker	Symbol	Subscribe this topic to get the realtime push of BBO changes. The ticker channel provides real-time price updates whenever a match happens. If multiple orders are matched at the same time, only the last matching event will be pushed.
SubscribeLevel2MarketData	Symbol	Subscribe this topic to get Level 2 order book data.
SubscribeExecutionData	Symbol	For each order executed, the system will send you the match messages in the format as following.
SubscribeLevel2_5BestAskBid	Symbol	Returned for every 100 milliseconds at most.
SubscribeLevel2_50BestAskBid	Symbol	Returned for every 100 milliseconds at most.
SubscribeContractMarketData	Symbol	Subscribe this topic to get the market data of the contract.
SubscribeSystemAnnouncements	Symbol	Subscribe this topic to get the system announcements.
SubscribeTransactionStatistics	Symbol	The transaction statistics will be pushed to users every 5 seconds.
SubscribeKlines	Symbol	Subscribe to contract klines (candlestick) data.
SubscribeFundingFeeSettlement	Symbol	Subscribe to funding fee settlement notifications.

If ACK parameter is sent to true, after a successful subscription / unsubscription, client receives a message about it.

Private Channels

Requires a valid ApiKey obtained from your Kucoin account. The ApiKey, ApiSecret and Passphrase must be set in the Kucoin property of the client API component.

The following data is pushed to client every time there is a change. There is no need to subscribe to any method, this is done automatically if you set a valid ApiKey.

Method	Description
SubscribeTradeOrders	This topic will push all change events of your orders.
SubscribeAccountBalance	You will receive this message when an account balance changes. The message contains the details of the change.
SubscribePositionChange	The system will push the change event when the position status changes.
SubscribeStopOrder	When a stop order is received by the system, you will receive a message with "open" type. It means that this order entered the system and waited to be triggered.
SubscribeMarginMode	Subscribe to margin mode changes. The system will push the change event when the margin mode is updated.

SubscribeCrossMargin-Leverage	Subscribe to cross margin leverage changes. The system will push the change event when the cross margin leverage is updated.

REST API

All endpoints return either a JSON object or array.

Public API EndPoints

These endpoints can be accessed without any authorization.

General EndPoints

Method	Parameters	Description
GetServiceSta-tus		Test connectivity to the Rest API and get the Service Status
GetServerTime		Test connectivity to the Rest API and get the current server time.

Market Data EndPoints

Method	Parameters	Description
GetOpenContractList		Submit request to get the info of all open contracts.
GetOrderInfoContract		Submit request to get info of the specified contract.
GetTicker	Symbol	The real-time ticker includes the last traded price, the last traded size, transaction ID, the side of liq-uidity taker, the best bid price and size, the best ask price and size as well as the transaction time of the orders. These messages can also be obtained through Websocket. The Sequence Number is used to judge whether the messages pushed by Web-socket is continuous.
GetPartOrderBook20	Symbol	Get a snapshot of aggregated open orders for a symbol.
GetPartOrderBook100	Symbol	Get a snapshot of aggregated open orders for a symbol.
GetFullOrderBook	Symbol	Get a snapshot of aggregated open orders for a symbol.
GetLevel2PullingMessages	Symbol	If the messages pushed by Websocket is not contin-uous, you can submit the following request and re-pull the data to ensure that the sequence is not missing. In the request, the start parameter is the sequence number of your last received message plus 1, and the end parameter is the sequence number of your current received message minus 1. After re-pulling the messages and applying them to your local exchange order book, you can continue to update the order book via Websocket incremen-

		tal feed. If the difference between the end and start parameter is more than 500, please stop using this request and we suggest you to rebuild the Level 2 orderbook.
GetTradeHistory	Symbol	List the last 100 trades for a symbol.
GetInterestRateList	Symbol	Check interest rate list.
GetIndexList	Symbol	Check index list
GetCurrentMarkPrice	Symbol	Check the current mark price.
GetPremiumIndex	Symbol	Submit request to get premium index.
GetCurrentFundingRate	Symbol	Submit request to check the current mark price.
GetKLine	Symbol	Get K Line Data of Contract

Private API EndPoints

Requires an APIKey and APISecret to get authorized by server.

User EndPoints

Method	Parameters	Description
GetAccoun-tOverview		Get Account Overview
GetTransac-tionHistory		If there are open positions, the status of the first page re-turned will be Pending, indicating the realised profit and loss in the current 8-hour settlement period. Please specify the minimum offset number of the current page into the offset field to turn the page.

Trade Endpoints

Method	Parameters	Description
PlaceOrder		You can place two types of orders: limit and mar-ket. Orders can only be placed if your account has sufficient funds. Once an order is placed, your funds will be put on hold for the duration of the order. The amount of funds on hold depends on the order type and parameters specified.
PlaceMarketOrder		Places a Market Order.
PlaceLimitOrder		Places a Limit Order.
CancelOrder		Cancels an Order by Order Id.
LimitOrderMassCancellation		Cancel all open orders (excluding stop orders). The response is a list of orderIDs of the canceled or-ders.
StopOrderMassCancellation		Cancel all untriggered stop orders. The response is a list of orderIDs of the canceled stop orders. To cancel triggered stop orders, please use 'Limit Or-der Mass Cancelation'.
GetOrderList		List your current orders.

GetUntriggeredStopOrderList		Get the un-triggered stop orders list.
GetListOrdersCompleted24hr		Get a list of recent 1000 orders in the last 24 hours. If you need to get your recent traded order history with low latency, you may query this end-point.
GetOrder		Get a single order by order id (including a stop or-der).
GetOrderByClientOid		Get a single order by client order id (including a stop order).
GetFills		Get a list of recent fills.
GetRecentFills		Get a list of recent 1000 fills in the last 24 hours. If you need to get your recent traded order history with low latency, you may query this endpoint.
ActiveOrderValueCalculation		You can query this endpoint to get the total num-ber and value of all your active orders.
GetPositionDetails		Get the position details of a specified position.
GetPositionList		Get the position details of a specified position.
AutoDepositMargin		Enable/Disable of Auto-Deposit Margin
AddMarginManually		Add Margin Manually
ObtainFuturesRiskLimitLevel		This interface can be used to obtain information about risk limit level of a specific contract
AdjustRiskLimitLevel		This interface is for the adjustment of the risk limit level. To adjust the level will cancel the open order, the response can only indicate whether the submit of the adjustment request is successful or not.
GetFundingHistory		Submit request to get the funding history.
GetMaxOpenSize		Get maximum open position size for a contract.
SwitchMarginMode		Switch between cross margin and isolated margin modes.
GetMarginMode		Get the current margin mode for a contract.

Events

Kucoin Messages are received in TsgcWebSocketClient component, you can use the following events:

OnConnect
After a successful connection to Kucoin server.
OnDisconnect
After a disconnection from Kucoin server
OnMessage
Messages sent by server to client are handled in this event.
OnError
If there is any error in protocol, this event will be called.
OnException
If there is an unhandled exception, this event will be called.

Additionally, there is a specific event in Kucoin API Component, called **OnKucoinHTTPException**, which is raised every time there is an error calling an HTTP Request (REST API or WebSocket Feeds).

Kucoin | Futures Connect WebSocket API

In order to connect to Kucoin WebSocket API, just create a new Kucoin API client and attach to TsgcWebSocket-Client.
See below an example:

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();

TsgcWSAPI_Kucoin_Futures

*oKucoin = new

TsgcWSAPI_Kucoin_Futures

();
oKucoin->Client = oClient;
oClient->Active = true;
```

Kucoin | Futures Subscribe WebSocket Channel

Kucoin offers a variety of channels where you can subscribe to get real-time updates of market data, orders... Find below a sample of how to subscribe to a Ticker:

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();

TsgcWSAPI_Kucoin_Futures

*oKucoin = new

TsgcWSAPI_Kucoin_Futures

();
oKucoin->Client = oClient;
oKucoin->SubscribeSymbolTickerV2("XBTUSDM");
void OnMessage(TsgcWSConnection *Connection, const string aText)
{
// here you will receive the ticker updates
}
```

Kucoin | Futures Get Market Data

Kucoin offers public Market Data through REST Endpoints, when you call one of these endpoints, you will get a snapshot of the market data requested.
The Market Data Endpoints don't require authentication, so are freely available to all users.

Example: to get the snapshot of the ticker BTC-USDT, do the following call

```
TsgcWSAPI_Kucoin_Futures

*oKucoin = new

TsgcWSAPI_Kucoin_Futures

(this);
ShowMessage(oKucoin->REST_API->GetTicker("

XBTUSDM

"));
```

Kucoin | Futures Private REST API

The Kucoin REST API offer public and private endpoints. The Private endpoints require that messages are signed to increase the security of transactions.
First you must login to your Kucoin account and create a new API, you will get the following values:

- ApiKey
- ApiSecret
- Passphrase

These fields must be configured in the Kucoin property of the Kucoin API client component.
Once configured, you can start to do private requests to the Kucoin Pro REST API

*Private Requests, require that your local machine has the local time synchronized, if not, the requests will be rejected by Kucoin server. Check the following article about this, Kucoin Private Requests Time.

```
TsgcWSAPI_Kucoin_Futures

*oKucoin = new

TsgcWSAPI_Kucoin_Futures

(this);
oKucoin->Kucoin->ApiKey = "<your api key>";
oKucoin->Kucoin->ApiSecret = "<your api secret>";
oKucoin->Kucoin->Passphrase = "<your passphrase>";
ShowMessage(oKucoin->REST_API->GetAccountOverview());
```

Kucoin | Futures Trade

Kucoin allows you to trade Futures using its REST API.

Configuration

First you must create an **API Key** in your Kucoin account and add privileges to trading with Futures.
Once this is done, you can start futures trading.
First, **set your ApiKey, ApiSecret and Passphrase** in the Kucoin Client Component, this will be used to sign the requests sent to Kucoin server.

Place an Order

To place a new order, just call to method **REST_API.PlaceOrder** of Kucoin Client Component.

Depending on the type of the order (market, limit...) the API requires more or less fields.

Parameters

Param	type	Description
clientOid	String	Unique order id created by users to identify their orders, e.g. UUID, Only allows numbers, characters, underline(_), and separator(-)
side	String	**buy** or **sell**
symbol	String	a valid contract code. e.g. XBTUSDM
type	String	*[optional]* Either **limit** or **market**
leverage	String	Leverage of the order
remark	String	*[optional]* remark for the order, length cannot exceed 100 utf8 characters
stop	String	*[optional]* Either **down** or **up**. Requires **stopPrice** and **stopPriceType** to be defined
stopPriceType	String	*[optional]* Either **TP**, **IP** or **MP**, Need to be defined if **stop** is specified.
stopPrice	String	*[optional]* Need to be defined if **stop** is specified.
reduceOnly	boolean	*[optional]* A mark to reduce the position size only. Set to false by default. Need to set the position size when reduceOnly is true.
closeOrder	boolean	*[optional]* A mark to close the position. Set to false by default. It will close all the positions when closeOrder is true.
forceHold	boolean	*[optional]* A mark to forcibly hold the funds for an order, even though it's an order to reduce the position size. This helps the order stay on the order book and not get canceled when the position size changes. Set to false by default.

LIMIT ORDER PARAMETERS

Param	type	Description
price	String	Limit price
size	Integer	Order size. Must be a positive number
timeIn-Force	String	*[optional]* **GTC**, **IOC**(default is **GTC**), read Time In Force
postOnly	boolean	*[optional]* Post only flag, invalid when **timeInForce** is **IOC**. When postOnly chose, not allowed choose hidden or iceberg.
hidden	boolean	*[optional]* Orders not displaying in order book. When hidden chose, not allowed choose postOnly.
iceberg	boolean	*[optional]* Only visible portion of the order is displayed in the order book. When iceberg chose, not allowed choose postOnly.
visibleSize	Integer	*[optional]* The maximum visible size of an iceberg order

MARKET ORDER PARAMETERS

Param	type	Description
size	Integer	*[optional]* amount of contract to buy or sell

When you send an order, there are 2 possibilities:

1. **Successful:** the function PlaceOrder returns the message sent by Kucoin server.
2. **Error:** the exception is returned in the event OnKucoinHTTPException.

Place Market Order 1 XBTUSDM

```
TsgcWSAPI_Kucoin_Futures

*oKucoin = new

TsgcWSAPI_Kucoin_Futures

(this);
oKucoin->Kucoin->ApiKey = "<api key>";
oKucoin->Kucoin->ApiSecret = "<api secret>";
oKucoin->Kucoin->Passphrase = "<passphrase>";
ShowMessage(oKucoin->REST_API->

PlaceMarketOrder

(

kosBuy, "XBTUSDM", 1

));
```

Place Limit Order 1 XBTUSDM at 40000

```
TsgcWSAPI_Kucoin_Futures
```

```
*oKucoin = new

TsgcWSAPI_Kucoin_Futures

(this);
oKucoin->Kucoin->ApiKey = "<api key>";
oKucoin->Kucoin->ApiSecret = "<api secret>";
oKucoin->Kucoin->Passphrase = "<passphrase>";
ShowMessage(oKucoin->REST_API->

PlaceLimitOrder

(

kosBuy, "XBTUSDM", 1, 40000

));
```

Kucoin | Futures Private Requests Time

When you do a private request to Kucoin, the message is signed to increase the security of requests. The message takes the local time and sends inside the signed message, if the local time has a difference greater than 5 seconds with Kucoin servers, the request will be rejected. So, it's important to verify that your local time is synchronized, you can do this using the synchronization time method for your OS.

You can check the Kucoin server time, calling method **GetServerTime**, which will return the time of the Kucoin server

API 3Commas

3Commas

APIs supported

- WebSockets API: connect to a public websocket server and provides real-time market data updates.
- REST API: The REST API has endpoints for account and order management as well as public market data.

WebSockets API

The websocket feed provides real-time market data updates for Trades and Deals

You can subscribe to the following **Public channels**:

Method	Arguments	Description
SubscribeSmartTrades		
SubscribeDeals		

These channels require **Authentication** against 3Commas servers. So first request your API keys in your 3Commas Account and then set the values in the property ThreeComas of the component:

- ApiKey
- ApiSecret

If the subscription is successful, the event **OnThreeCommasConfirmSubscription** will be called. If not, the event **OnThreeCommasRejectSubscription** is called, you can get the reason of the rejection using the **aRawMessage** parameter.

REST API

Test Connectivity

Method	Arguments	Description
GetPing		
GetServerTime		Returns the server time

Account

Method	Arguments	Description
GetAccounts		User connected exchanges list
GetMarketList		Supported Market List
GetMarketPairs	**aMarketCode:** code of the market	All market pairs
GetCurrencyRatesWithLeverageData	**aMarketCode:** code of the market **aPair:** pair name	Currency rates and limits with leverage data
GetCurrencyRates	**aMarketCode:** code of the market **aPair:** pair name	Currency rates and limits
GetBalances	**aAccountId:** id of the account	Load balances for specified exchange
GetAccountTableData	**aAccountId:** id of the account	Information about all user balances on specified exchange
GetAccountLeverage	**aAccountId:** id of the account **aPair:** pair name	Information about account leverage
GetAccountInfo	**aAccountId:** id of the account	Single Account Info

Smart Trades

Method	Arguments	Description
GetSmartTradeHistory		Get the Trade History
PlaceMarketOrder	**aAccountId:** id of the account **aOrderSide:** buy or sell **aPair:** pair name **aQuantity:** amount	Places a Market Order
PlaceLimitOrder	**aAccountId:** id of the account **aOrderSide:** buy or sell **aPair:** pair name **aQuantity:** amount **aPrice:** limit price	Places a Limit Order
GetSmartTrade	**aId:** id of the trade	Get a Smart Trade by the Id of the Trade

CancelSmartTrade	**aId:** id of the trade	Cancel a Smart Trade by the Id of the Trade
CloseByMarketSmartTrade	**aId:** id of the trade	
EditSmartTrade	**aId:** id of the trade	Edit an existing Smart Trade
ForceStartSmartTrade	**aId:** id of the trade	Force start a Smart Trade
AddFundsSmartTrade	**aId:** id of the trade	Add funds to a Smart Trade
GetSmartTradeTrades	**aId:** id of the trade	Get trades of a Smart Trade

DCA Bot

Method	Arguments	Description
CreateDCABot		Create a new DCA Bot
GetDCABot	**aId:** id of the bot	Get a DCA Bot by Id
GetDCABots		Get all DCA Bots
EnableDCABot	**aId:** id of the bot	Enable a DCA Bot
DisableDCABot	**aId:** id of the bot	Disable a DCA Bot
DeleteDCABot	**aId:** id of the bot	Delete a DCA Bot
CancelDCABot	**aId:** id of the bot	Cancel a DCA Bot
GetDCABotStats		Get DCA Bot statistics
GetAvailableStrategyList		Get available strategy list
GetBlacklistPairs		Get blacklist pairs
AddBlacklistPairs		Add blacklist pairs

Deals

Method	Arguments	Description
GetDeals		Get all deals
GetDeal	**aId:** id of the deal	Get a deal by Id

UpdateDeal	**aId:** id of the deal	Update a deal
CancelDeal	**aId:** id of the deal	Cancel a deal
CloseAtMarketDeal	**aId:** id of the deal	Close a deal at market price

Grid Bot

Method	Arguments	Description
CreateGridBot		Create a new Grid Bot
GetGridBot	**aId:** id of the bot	Get a Grid Bot by Id
GetGridBots		Get all Grid Bots
EnableGridBot	**aId:** id of the bot	Enable a Grid Bot
DisableGridBot	**aId:** id of the bot	Disable a Grid Bot
DeleteGridBot	**aId:** id of the bot	Delete a Grid Bot

Events

OnConnect

When a new WebSocket connection is open

OnDisconnect

When a WebSocket connection is closed

OnThreeCommasConnect

When the client receives a Welcome message from 3Commas server, means the connection is ready.

OnThreeCommasConfirmSubscription

Confirms a previous subscription sent by the client.

OnThreeCommasRejectSubscription

There is an error trying to subscribe to a 3Commas channel

OnThreeCommasMessage

Here the client receives the data sent by server related to the channels subscribed

OnThreeCommasPing

Ping sent by server to the client.

OnThreeCommasHTTPException

If there is any error while calling HTTP REST methods, this event will be called.

API OKX

OKX

APIs supported

- WebSockets API: connect to a websocket server and provides real-time market data updates, account changes and place trading orders.

Properties

WebSocket channels are divided into two categories: public and private channels.

- **Public channels:** include tickers channel, K-Line channel, limit price channel, order book channel, and mark price channel, etc -- do not require log in.
- **Private channels:** including account channel, order channel, and position channel, etc -- require log in.

You can configure the following properties in the OKX property.

- **ApiKey:** you can request a new api key in your OKX account, just copy the value to this property.
- **ApiSecret:** it's the secret value of the api.
- **Passphrase:** it's the custom string defined when creating a new api key.
- **IsDemo:** if enabled, will connect to the OKX Demo account (disabled by default).
- **IsPrivate:** if enabled, you will be able to connect to private channels (disabled by default).
- **IsBusiness:** if enabled, will connect to the Business WebSocket endpoint (wss://ws.okx.com:8443/ws/v5/business). Use this for candle channels, algo order channels, and other advanced channels (disabled by default).
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false). The replay is sent as combined frames, so it does not spend one of the 480 operations OKX allows per hour for every channel, read Batch Subscriptions below.

Connection

When the client successfully connects to OKX servers, the event **OnOKXConnect** is fired. If there is any error while trying to connect, the event **OnOKXError** will be fired with the error details.
After the event **OnOKXConnect** is fired, then you can start to **send** and **receive messages** from OKX servers.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_OKX *oOKX = new TsgcWSAPI_OKX(NULL);
oOKX->Client = oClient;
oOKX->OKX->ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oOKX->OKX->ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oOKX->OKX->Passphrase = "secret_passphrase";
oClient->Active = true;
void OnOKXConnect(TObject *Sender, string aMessage, string aCode, string aRawMessage)
{
  DoLog("#OKX Connected");
}
void OnOKXError(TObject *Sender, string aCode, string aMessage, string aRawMessage)
{
  DoLog("#error: " + aMessage);
}
```

Public Channels

The websocket feed provides real-time market data updates for orders and trades. The websocket feed has some public channels like ticker, trades...

You can subscribe to the following **Public channels**:

Method	Description
SubscribeInstruments	The full instrument list will be pushed for the first time after subscription. Subsequently, the instruments will be pushed if there is any change to the instrument's state (such as delivery of FUTURES, exercise of OPTION, listing of new contracts / trading pairs, trading suspension, etc.).
SubscribeTicker	Retrieve the last traded price, bid price, ask price and 24-hour trading volume of instruments. Data will be pushed every 100 ms.
SubscribeOpenInterest	Retrieve the open interest. Data will be pushed every 3 seconds.
SubscribeCandlestick	Retrieve the candlesticks data of an instrument. the push frequency is the fastest interval 500ms push the data.
SubscribeTrades	Retrieve the recent trades data. Data will be pushed whenever there is a trade.
SubscribeEstimated-Prices	Retrieve the estimated delivery/exercise price of FUTURES contracts and OPTION. Only the estimated delivery/exercise price will be pushed an hour before delivery/exercise, and will be pushed if there is any price change.
SubscribeMarkPrice	Retrieve the mark price. Data will be pushed every 200 ms when the mark price changes, and will be pushed every 10 seconds when the mark price does not change.
SubscribeMarkPrice-Candlestick	Retrieve the candlesticks data of the mark price. Data will be pushed every 500 ms.
SubscribePriceLimit	Retrieve the maximum buy price and minimum sell price of the instrument. Data will be pushed every 5 seconds when there are changes in limits, and will not be pushed when there is no changes on limit.
SubscribeOrderBook	Retrieve order book data. Use books for 400 depth levels, book5 for 5 depth levels, bbo-tbt tick-by-tick 1 depth level, books50-l2-tbt tick-by-tick 50 depth levels, and books-l2-tbt for tick-by-tick 400 depth levels. • **books:** 400 depth levels will be pushed in the initial full snapshot. Incremental data will be pushed every 100 ms when there is change in order book. • **books5:** 5 depth levels will be pushed every time. Data will be pushed every 100 ms when there is change in order book. • **bbo-tbt**: 1 depth level will be pushed every time. Data will be pushed every 10 ms when there is change in order book. • **books-l2-tbt:** 400 depth levels will be pushed in the initial full snapshot. Incremental data will be pushed every 10 ms when there is change in order book. • **books50-l2-tbt:** 50 depth levels will be pushed in the initial full snapshot. Incremental data will be pushed every 10 ms when there is change in order book. If asks or bids is an empty array, it means that there are changes in 400 depth, instead of 50 depth. If you maintain the order book data locally, please ignore empty asks and bids.
SubscribeOptionSummary	Retrieve detailed pricing information of all OPTION contracts. Data will be pushed at once.
SubscribeFundingRate	Retrieve funding rate. Data will be pushed in 30s to 90s.
SubscribeIndexCandlestick	Retrieve the candlesticks data of the index. Data will be pushed every 500 ms.
SubscribeIndexTicker	Retrieve index tickers data

SubscribeStatus	Get the status of system maintenance and push when the system maintenance status changes. First subscription: "Push the latest change data"; every time there is a state change, push the changed content
SubscribePublicStructureBlockTrades	Data will be pushed whenever there is a block trade.
SubscribeBlockTickers	Retrieve the latest block trading volume in the last 24 hours. The data will be pushed when triggered by transaction execution event. In addition, it will also be pushed in 5 minutes interval according to subscription granularity.
SubscribeAllTrades	Retrieve all trades data. Data will be pushed whenever there is a trade.
SubscribeLiquidationOrders	Retrieve liquidation orders. Data will be pushed when there is a liquidation order.
SubscribeADLWarning	Retrieve ADL warning data. Data will be pushed when the auto-deleveraging risk increases.
SubscribeEconomicCalendar	Retrieve the economic calendar events. Data will be pushed when there are updates to economic events.
SubscribePublicBlockTrades	Retrieve public block trades. Data will be pushed whenever there is a block trade.
SubscribeOptionTrades	Retrieve option trades data. Data will be pushed whenever there is an option trade.
SubscribeCallAuctionDetails	Retrieve call auction details. Data will be pushed when there are updates to call auction information.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_OKX *oOKX = new TsgcWSAPI_OKX(NULL);
oOKX->Client = oClient;
oOKX->OKX->ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oOKX->OKX->ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oOKX->OKX->Passphrase = "secret_passphrase";
oClient->Active = true;
void OnOKXConnect(TObject *Sender, string aMessage, string aCode, string aRawMessage)
{
  oOKX->SubscribeInstruments(okxitFutures);
}
```

Private Channels

Including account channel, order channel, and position channel, etc -- require log in.

You can subscribe to the following **Private channels**:

Method	Description
SubscribeAccount	Retrieve account information. Data will be pushed when triggered by events such as placing order, canceling order, transaction execution, etc. It will also be pushed in regular interval according to subscription granularity.
SubscribePositions	Retrieve position information. Initial snapshot will be pushed according to subscription granularity. Data will be pushed when triggered by events such as placing/canceling order, and will also be pushed in regular interval according to subscription granularity.
SubscribeBalanceAndPosition	Retrieve account balance and position information. Data will be pushed when triggered by events such as filled order, funding transfer.
SubscribeOrders	Retrieve order information. Data will not be pushed when first subscribed. Data will only be pushed when triggered by events such as placing/canceling order.

SubscribeOrdersAlgo	Retrieve algo orders (includes trigger order, oco order, conditional order). Data will not be pushed when first subscribed. Data will only be pushed when triggered by events such as placing/canceling order.
SubscribeAdvanceAlgo	Retrieve advance algo orders (including Iceberg order, TWAP order, Trailing order). Data will be pushed when first subscribed. Data will be pushed when triggered by events such as placing/canceling order.
SubscribePositionRisk	This push channel is only used as a risk warning, and is not recommended as a risk judgment for strategic trading In the case that the market is not moving violently, there may be the possibility that the position has been liquidated at the same time that this message is pushed.
SubscribeAccount-Greeks	Retrieve account greeks information. Data will be pushed when triggered by events such as increase/decrease positions or cash balance in account, and will also be pushed in regular interval according to subscription granularity.
SubscribeRfqs	Retrieve the Rfqs.
SubscribeQuotes	Retrieve the Quotes.
SubscribePrivat-eStructureBlock-Trades	Retrieve Structure Block Trades.
SubscribeSpotGridAl-goOrders	Retrieve spot grid algo orders. Data will be pushed when first subscribed. Data will be pushed when triggered by events such as placing/canceling order.
SubscribeContactGri-dAlgoOrders	Retrieve contract grid algo orders. Data will be pushed when first subscribed. Data will be pushed when triggered by events such as placing/canceling order.
SubscribeGridPosi-tions	Retrieve grid positions. Data will be pushed when first subscribed. Data will be pushed when triggered by events such as placing/canceling order.
SubscribeGridSubOr-ders	Retrieve grid sub orders. Data will be pushed when first subscribed. Data will be pushed when triggered by events such as placing order.
SubscribeFills	Retrieve filled orders data. Data will be pushed when an order is filled.
SubscribeDepositInfo	Retrieve deposit information. Data will be pushed when there is a deposit status update.
SubscribeWithdrawal-Info	Retrieve withdrawal information. Data will be pushed when there is a withdrawal status update.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_OKX *oOKX = new TsgcWSAPI_OKX(NULL);
oOKX->Client = oClient;
oOKX->OKX->ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oOKX->OKX->ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oOKX->OKX->Passphrase = "secret_passphrase";
oClient->Active = true;
void OnOKXConnect(TObject *Sender, string aMessage, string aCode, string aRawMessage)
{
  oOKX->SubscribeOrders(okxitFutures, "BTC-USD", "BTC-USD-200329");
}
```

Batch Subscriptions

OKX limits a connection to 480 subscribe, unsubscribe and login operations per hour. Subscribing many channels one at a time, or replaying a large watchlist after a reconnection, can exhaust that budget after only a few reconnections.

Method	Description
BeginBatch	Opens a batch. Every Subscribe and UnSubscribe call made until the matching EndBatch is buffered instead of sent immediately. Calls can be nested; close

	each one in a finally block, the batch is only written when the outermost EndBatch is reached.
EndBatch	Closes the batch opened by BeginBatch and writes the buffered channels, one frame per run of Subscribe or UnSubscribe calls and never larger than the 64 KB OKX documents for the channels of a request.
SubscribeChannels	Subscribes to a list of channel names with a single frame. Pass the instrument id when the channels need one.
UnSubscribeChannels	Unsubscribes from a list of channel names with a single frame. Pass the instrument id when the channels need one.

Group any of the Subscribe and UnSubscribe methods above into a single frame with **BeginBatch** and **EndBatch**:

```
oOKX->BeginBatch();
try
{
  oOKX->SubscribeTrades("BTC-USDT");
  oOKX->SubscribeOrderBook("BTC-USDT", okxobBooks5);
  oOKX->SubscribeMarkPrice("BTC-USDT");
}
__finally
{
  oOKX->EndBatch();
}
```

For a list of channel names, **SubscribeChannels** and **UnSubscribeChannels** do the same in one call:

```
oOKX->SubscribeChannels(ARRAYOFCONST(("trades", "books5", "tickers")), "BTC-USDT");

// and to unsubscribe
oOKX->UnSubscribeChannels(ARRAYOFCONST(("trades", "books5", "tickers")), "BTC-USDT");
```

When **Resubscribe** is enabled, the channels active before a reconnection are replayed the same way, as combined frames, with no code needed.

Trading

The WebSocket Trade requires **Authentication**.

You can place an order only if you have sufficient funds. Find below a table with the request parameters:

Parameter	Type	Required	Description
id	String	Yes	Unique identifier of the message Provided by client. It will be returned in response message for identifying the corresponding request. A combination of case-sensitive alphanumerics, all numbers, or all letters of up to 32 characters.
> instId	String	Yes	Instrument ID,e.g. `BTC-USD-190927-5000-C`
> tdMode	String	Yes	Trade mode Margin mode `isolated` `cross` Non-Margin mode `cash`
> ccy	String	No	Margin currency Only applicable to `cross` `MARGIN` orders in `Single-currency margin`.
> clOrdId	String	No	Client-supplied order ID A combination of case-sensitive alphanumerics, all numbers, or all letters of up to 32 characters.

Parameter	Type	Required	Description
> tag	String	No	Order tag A combination of case-sensitive alphanumerics, all numbers, or all letters of up to 16 characters.
> side	String	Yes	Order side, `buy` `sell`
> posSide	String	Optional	Position side The default is `net` in the `net` mode It is required in the `long/short` mode, and can only be `long` or `short`. Only applicable to `FUTURES/SWAP`.
> ordType	String	Yes	Order type `market`: market order `limit`: limit order `post_only`: Post-only order `fok`: Fill-or-kill order `ioc`: Immediate-or-cancel order `optimal_limit_ioc`:Market order with immediate-or-cancel order
> sz	String	Yes	Quantity to buy or sell.
> px	String	Optional	Price Only applicable to `limit`, `post_only`, `fok`, `ioc` order.
> reduceOnly	Boolean	No	Whether to reduce position only or not, `true` `false`, the default is `false`. Only applicable to `MARGIN` orders, and `FUTURES/SWAP` orders in `net` mode Only applicable to `Single-currency margin` and `Multi-currency margin`
> tgtCcy	String	No	Quantity type `base_ccy`: Base currency, `quote_ccy`: Quote currency Only applicable to `SPOT` traded with Market order Default is `quote_ccy` for buy, `base_ccy` for sell
> banAmend	Boolean	No	Whether to ban amending spot order or not, true or false, the default is false. It will fail to place orders if the balance is not enough when banAmend is true. Only applicable to SPOT market order

Place Order Example

You can place an order only if you have sufficient funds.

```
// Place Martket Order
TsgcWSAPI_OKX1->PlaceMarketOrder(okxosBuy, "ETH-BTC", 1);
// Place Limit Order
TsgcWSAPI_OKX1->PlaceLimitOrder(okxosBuy, "ETH-BTC", 1, 0.25);
```

Cancel Order Example

Cancel an incomplete order

```
TsgcWSAPI_OKX1->CancelOrder("
ETH-BTC
```

```
", "457589362405027840");
```

Amend Order

Amend an incomplete order.

```
TsgcWSAPI_OKX1->AmendOrder("ETH-BTC", "457589362405027840", "", 2);
```

Batch Trade Operations

The WebSocket Trade API also supports batch operations for placing, canceling, and amending multiple orders at once. These operations require **Authentication**.

Method	Description
BatchPlaceOrders	Place multiple orders in a single request. Maximum 20 orders can be placed at a time.
BatchCancelOrders	Cancel multiple orders in a single request. Maximum 20 orders can be canceled at a time.
BatchAmendOrders	Amend multiple incomplete orders in a single request. Maximum 20 orders can be amended at a time.
MassCancelOrders	Mass cancel all pending orders for a specific instrument type.

API XTB

XTB

APIs supported

- WebSockets API: connect to a websocket server and provides real-time market data updates, account changes and place trading orders.

Properties

The WebSocket protocol allows 2 types of requests: **Streaming commands** (receive live updates) and **Retrieve Trading Data** (send a request to server retrieving some information).

You can configure the following properties in the XTB property.

- **User:** the username that identifies the connection.
- **Password:** it's the secret value of the user.
- **Demo:** if enabled, will connect to the XTB Demo account (disabled by default).
- **Resubscribe:** if enabled, after a reconnection and a successful login the client automatically subscribes again to the streaming subscriptions that were active before the disconnection (by default false).

Connection

When the client successfully connects to XTB servers, the event **OnXTBConnect** is fired. If there is any error while trying to connect, the event **OnXTBError** will be fired with the error details.
After the event **OnXTBConnect** is fired, then you can start to **send** and **receive messages** from XTB servers.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_XTB *oXTB = new TsgcWSAPI_XTB(NULL);
oXTB->Client = oClient;
oXTB->XTB->User = "user_0001";
oXTB->XTB->Password = "secret_0001";
oClient->Active = true;
void OnXTBConnect(TObject *Sender, string aStreamSessionId)
{
  DoLog("#XTB Connected");
}
void OnXTBError(TObject *Sender, string aCode, string aDescription, string aRawMessage)
{
  DoLog("#error: " + aDescription);
}
```

Connection Commands

Method	Description
Login	In order to perform any action client application have to perform login process. No functionality is available before proper login process. The login method is called automatically after the client connects to the websocket server and the User/Password values are set.
Logout	

Streaming Commands

You can subscribe to the following channels:

Method	Description
SubscribeBalance	Allows to get actual account indicators values in real-time, as soon as they are available in the system.
SubscribeCandles	Subscribes for and unsubscribes from API chart candles. The interval of every candle is 1 minute. A new candle arrives every minute.
SubscribeKeepAlive	Subscribes for and unsubscribes from 'keep alive' messages. A new 'keep alive' message is sent by the API every 3 seconds.
SubscribeNews	Subscribes for and unsubscribes from news.
SubscribeProfits	Subscribes for and unsubscribes from profits.
SubscribeTickPrices	Establishes subscription for quotations and allows you to obtain the relevant information in real-time, as soon as it is available in the system. The getTickPrices command can be invoked many times for the same symbol, but only one subscription for a given symbol will be created. Please beware that when multiple records are available, the order in which they are received is not guaranteed.
SubscribeTrades	Establishes subscription for user trade status data and allows you to obtain the relevant information in real-time, as soon as it is available in the system. Please beware that when multiple records are available, the order in which they are received is not guaranteed.
SubscribeTradeStatus	Allows to get status for sent trade requests in real-time, as soon as it is available in the system. Please beware that when multiple records are available, the order in which they are received is not guaranteed
SubscribePing	Regularly calling this function is enough to refresh the internal state of all the components in the system. Streaming connection, when any command is not sent by client in the session, generates only one way network traffic. It is recommended that any application that does not execute other commands, should call this command at least once every 10 minutes.

Retrieving Trading Data

You can send the following Requests:

Method	Description
GetAllSymbols	Returns array of all symbols available for the user.
GetCalendar	Returns calendar with market events.
GetChartLastRequest	Please note that this function can be usually replaced by its streaming equivalent getCandles which is the preferred way of retrieving current candle data. Returns chart info, from start date to the current time. If the chosen period of CHART_LAST_INFO_RECORD is greater than 1 minute, the last candle returned by the API can change until the end of the period (the candle is being automatically updated every minute).

GetChartRangeRequest	Please note that this function can be usually replaced by its streaming equivalent getCandles which is the preferred way of retrieving current candle data. Returns chart info with data between given start and end dates.
GetCommissionDef	Returns calculation of commission and rate of exchange. The value is calculated as expected value, and therefore might not be perfectly accurate.
GetCurrentUserData	Returns information about account currency, and account leverage.
GetIbsHistory	Returns IBs data from the given time range.
GetMarginLevel	Please note that this function can be usually replaced by its streaming equivalent getBalance which is the preferred way of retrieving account indicators. Returns various account indicators
GetMarginTrade	Returns expected margin for given instrument and volume. The value is calculated as expected margin value, and therefore might not be perfectly accurate.
GetNews	Please note that this function can be usually replaced by its streaming equivalent getNews which is the preferred way of retrieving news data. Returns news from trading server which were sent within specified period of time.
GetProfitCalculation	Calculates estimated profit for given deal data Should be used for calculator-like apps only. Profit for opened transactions should be taken from server, due to higher precision of server calculation
GetServerTime	Returns current time on trading server
GetStepRules	Returns a list of step rules for DMAs
GetSymbol	Returns information about symbol available for the user
GetTickPrices	Please note that this function can be usually replaced by its streaming equivalent getTickPrices which is the preferred way of retrieving ticks data. Returns array of current quotations for given symbols, only quotations that changed from given timestamp are returned. New timestamp obtained from output will be used as an argument of the next call of this command.
GetTradeRecords	Returns array of trades listed in orders argument
GetTrades	Please note that this function can be usually replaced by its streaming equivalent getTrades which is the preferred way of retrieving trades data. Returns array of user's trades.
GetTradesHistory	Please note that this function can be usually replaced by its streaming equivalent getTrades which is the preferred way of retrieving trades data. Returns array of user's trades which were closed within specified period of time.
GetTradingHours	Returns quotes and trading times.
GetVersion	Returns the current API version.
Ping	Regularly calling this function is enough to refresh the internal state of all the components in the system. It is recommended that any application that does not execute other commands, should call this command at least once every 10 minutes. Please note that the streaming counterpart of this function is combination of ping and getKeepAlive
TradeTransaction	Starts trade transaction. tradeTransaction sends main transaction information to the server.
TradeTransactionStatus	Please note that this function can be usually replaced by its streaming equivalent getTradeStatus which is the preferred way of retrieving transaction status data. Returns current transaction status. At any time of transaction processing client might check the status of transaction on server side. In order to do that client must provide unique order taken from tradeTransaction invocation.

API Bybit

Bybit

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Currently, the API supported version is V5. The V5 API brings uniformity and efficiency to Bybit's product lines, unifying Spot, Derivatives, and Options in one set of specifications.

OpenAPI Version	Account Type	Linear			Inverse		Spot	Options
		USDT Perpetual	**USDC Perpetual**	**USDC Futures**	**Perpetual**	**Futures**		
V5	Unified trading account	✓	✓	✓	see note		✓	✓
	Classic account	✓			✓	✓	✓	
V3	Unified trading account	✓	✓					✓
	Classic account	✓			✓	✓	✓	

*Note: the Unified account supports inverse trading. However, the margin used is from the inverse derivatives wallet instead of the unified wallet.

Properties

You can configure the following properties in the Bybit property.

- **ApiKey:** you can request a new api key in your Bybit account, just copy the value to this property. If the APIKey is set, the client will connect to the websocket private server. If it's empty, will connect to the WebSocket public server.
- **ApiSecret:** it's the secret value of the api.
- **SignatureExpires:** number of seconds after the signature expires (by default 10 seconds).
- **TestNet:** if enabled, will connect to the Bybit TestNet Demo account (disabled by default).
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Connection

When the client successfully connects to Bybit servers, the event **OnConnect** is fired. After the event **OnConnect** is fired, then you can start to **send** and **receive messages** to/from Bybit servers. If you are connecting to the private websocket channel, you must wait till **OnBybitAuthentication** event is fired and check if the success parameter is true, before subscribing to any channel.

The client supports several APIs, so use the property BybitClient to set which API you want to use:

- **bybSpot**
- **bybInverse**
- **bybLinear**
- **bybPerpetual**

Find below an example of connecting to WebSocket Spot Private API.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bybit *oBybit = new TsgcWSAPI_Bybit(NULL);
oBybit->Client = oClient;
oBybit->Bybit->ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oBybit->Bybit->ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oBybit->BybitClient = bybSpot;
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
  DoLog("#Bybit Connected");
}
```

After a successful connection to the Spot WebSocket Server, you can start to subscribe to WebSocket channels, just access the **SPOT** property and then call any of the subscribe/unsubscribe methods available.

Events

The bybit client implements the following events to control the connection flow and get data sent from the WebSocket server:

- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnBybitAuthentication:** fired when the client authenticates against the Private WebSocket Server.
- **OnBybitSubscribe:** when the client subscribes to a websocket channel.
- **OnBybitUnSubscribe:** when the client unsubscribes from a websocket channel.
- **OnBybitData:** when the client receives data from the server.
- **OnBybitError:** when there is any error during the bybit websocket connection.
- **OnBybitHTTPException:** when there is any error during the REST request.

WebSocket API

The websocket feed provides real-time market data updates for orders and trades. The websocket feed has some public channels like ticker, trades...

You can subscribe to the following **channels**:

Method	Public or Private	Description
SubscribeOrderBook	Public	Subscribe to the orderbook stream. Supports different depths.
SubscribeTrade	Public	Subscribe to the recent trades stream.
SubscribeTicker	Public	Subscribe to the ticker stream.

SubscribeKLine	Public	Subscribe to the klines stream.
SubscribeLiquidation	Public	Subscribe to the liquidation stream
SubscribeLT_KLine	Public	Subscribe to the leveraged token kline stream.
SubscribeLT_Ticker	Public	Subscribe to the leveraged token ticker stream.
SubscribeLT_Nav	Public	Subscribe to the leveraged token ticker stream.
SubscribePosition	Private	Subscribe to the leveraged token nav stream.
SubscribeExecution	Private	Subscribe
SubscribeOrder	Private	Subscribe
SubscribeWallet	Private	Subscribe
SubscribeGreek	Private	Subscribe
SubscribeDcp	Private	Subscribe
SubscribeInsurance	Public	Subscribe to the insurance fund stream.
SubscribeOrderPriceLimit	Public	Subscribe to the order price limit stream.
SubscribeADLAlert	Public	Subscribe to the auto-deleverage alert stream.
SubscribeFastExecution	Private	Subscribe to the fast execution stream.

Find below an example of subscribing to private websocket channels after a successful authentication.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bybit *oBybit = new TsgcWSAPI_Bybit(NULL);
oBybit->Client = oClient;
oBybit->Bybit->ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oBybit->Bybit->ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oBybit->BybitClient = bybSpot;
oClient->Active = true;
void OnBybitAuthentication(TObject *Sender, bool aSuccess, const string aError, const string aRawMessage)
{
  if (aSuccess == true)
  {
    oClient->SubscribeOrderBook("BTCUSDT");
    oClient->SubscribeTrade("BTCUSDT");
  }
}
```

REST API

The REST API has a list of Public and Private methods to request data from: markets, private account and wallet. Find below a list of available methods.

Method	Public / Private
GetServerTime	Public
GetKLine	Public
GetMarkPriceKLine	Public
GetIndexPriceKLine	Public
GetPremiumIndexPriceKLine	Public
GetInstrumentsInfo	Public
GetOrderBook	Public
GetTickers	Public
GetFundingRateHistory	Public
GetPublicRecentTradingHistory	Public
GetOpenInterest	Public
GetHistoricalVolatility	Public
GetInsurance	Public
GetRiskLimit	Public
GetDeliveryPrice	Public

GetLongShortRatio	Public
PlaceOrder	Private
PlaceMarketOrder	Private
PlaceLimitOrder	Private
AmendOrder	Private
CancelOrder	Private
GetOpenOrders	Private
CancelAllOrders	Private
GetOrderHistory	Private
GetPositionInfo	Private
SetLeverage	Private
SwitchCrossIsolatedMargin	Private
SetTPSLMode	Private
SwitchPositionMode	Private
SetRiskLimit	Private
SetTradingStop	Private
SetAutoAddMargin	Private
AddOrReduceMargin	Private
GetExecution	Private
GetClosedPNL	Private
ConfirmNewRiskLimit	Private
GetWalletBalance	Private
GetAccountInfo	Private
GetTransactionLog	Private
BatchPlaceOrder	Private
BatchAmendOrder	Private
BatchCancelOrder	Private
SetDCP	Private
GetFeeRate	Private
GetCollateralInfo	Private
SetMarginMode	Private
GetBorrowHistory	Private
GetCoinGreeks	Private
GetCoinInfo	Private
GetAllCoinsBalance	Private
CreateInternalTransfer	Private
GetInternalTransferList	Private
GetDepositRecords	Private
GetDepositAddress	Private
CreateWithdrawal	Private
CancelWithdrawal	Private
GetWithdrawalRecords	Private

Find below an example of getting the open orders.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bybit *oBybit = new TsgcWSAPI_Bybit(NULL);
oBybit->Client = oClient;
oBybit->Bybit->ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oBybit->Bybit->ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oBybit->BybitClient = bybSpot;
```

```
oBybit->REST_API->GetAccountInfo();
```

API Forex

Forex.com (StoneX / CityIndex)
Forex.com is a retail trading venue operated by StoneX. The component wraps the Trading REST API and the Lightstreamer-based streaming service used by Forex.com, StoneX and CityIndex (all three share the same platform). The following APIs are supported:

1. **Streaming API:** Lightstreamer TLCP 2.5 over WebSocket. Price ticks, order updates, position updates, account margin and quotes are pushed in real time after a successful REST LogOn.
2. **REST API:** session login (UserName + Password + AppKey), trade and stop-limit order management, account queries and market data. Uses HTTPS.

Components

- **TsgcWSAPI_Forex:** main client. Holds **Credentials**, exposes **REST_API**, the internal **Lightstreamer** streaming client, and the **OnForex*** events.
- **TsgcHTTP_API_Forex:** REST component, reachable through the **REST_API** property of the main client.
- **TsgcWSPClient_Lightstreamer:** internal TLCP 2.5 subprotocol client used for streaming subscriptions (PRICES / ORDERS / QUOTES / CLIENTACCOUNTMARGIN / TRADEMARGIN).

Published Properties

- **Credentials:** group of fields required to log in to the REST API.
  - **UserName:** your Forex.com / StoneX / CityIndex account user name.
  - **Password:** account password.
  - **AppKey:** application key issued by the venue.
  - **AppVersion:** optional client version string.
  - **AppComments:** free-form text logged on login.
  - **StreamingHost:** Lightstreamer host (defaults to the public streaming host).
- **AutoReconnect:** when enabled, forwards to the underlying WebSocket client WatchDog so that the streaming session reconnects automatically after a network drop (by default true).
- **REST_API:** reference to the **TsgcHTTP_API_Forex** component.
- **Lightstreamer:** reference to the internal **TsgcWSPClient_Lightstreamer** subprotocol client (AdapterSet is fixed at STREAMINGALL).
- **SessionToken:** read-only. Session token returned by the REST LogOn call; used as *LS_password* for Lightstreamer.
- **ClientAccountId / TradingAccountId:** read-only. Populated after a successful LogOn.

Published Events

OnForexConnect
Fired once the streaming session has been created and subscriptions can start.
OnForexDisconnect
Fired when the streaming session ends (server close, network drop or explicit Disconnect).
OnForexException
Raised on REST errors (including LogOn failure) and on streaming session faults.
OnForexPriceTick
Fired for every PRICES update: Bid, Offer, Price, Change, Direction, AuditId, TickDate.
OnForexOrderUpdate
Fired for every ORDERS update (new, filled, cancelled, rejected).
OnForexPositionUpdate
Fired for every open-position update on the active TradingAccount.
OnForexAccountMargin
Fired for every CLIENTACCOUNTMARGIN / TRADEMARGIN update (equity, margin, free margin, currency).
OnForexQuote
Fired for every QUOTES update (request-for-quote workflow).
OnForexSubscribe

Fired each time the server confirms a Lightstreamer subscription (SUBOK / SUBCMD). Parameters: *aAdapter* (PRICES, ORDERS, QUOTES, CLIENTACCOUNTMARGIN or TRADEMARGIN), *aGroup* (item group, e.g. *ID.<ClientAccountId>*), *aItems*, *aFields*.

OnForexUnsubscribe

Fired when a subscription ends — either via **UnwatchMarket**, on streaming disconnect, or when the server sends an UNSUB frame. Parameters: *aAdapter*, *aGroup*.

Most common uses

- **REST API**
  - **How to Connect to REST API**
  - **How to Place Trade and Stop-Limit Orders**
  - **How to Query Accounts, Positions and Order History**
  - **How to Get Market Data**
- **WebSockets API**
  - **How to Connect to WebSocket API**
  - **How to Subscribe to Streaming Adapters**

Sample

Create the client, set credentials, connect, subscribe to a market price stream and disconnect.

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey  = "<appkey>";
oForex->OnForexPriceTick = ForexPriceTick;
oForex->Connect();
oForex->WatchMarket(401484830);
oForex->Disconnect();
```

Forex | Connect REST API

Every call to the Forex.com / StoneX / CityIndex Trading API requires an active session. Session state is obtained with **LogOn**, ended with **LogOff**, and kept alive with **Ping**.

Credentials

Set the credentials on the main client component. They are copied into the REST session on the first call.

- **UserName:** Forex.com / StoneX / CityIndex account user name.
- **Password:** account password.
- **AppKey:** application key issued by the venue.
- **AppVersion / AppComments:** optional strings logged on login.

Session Methods

Method	Parameters	Description
LogOn	UserName, Password, AppKey	Opens a REST session. Returns the full JSON response and stores SessionToken, ClientAccountId and TradingAccountId on the main client. All later REST and streaming calls authenticate with that token.
LogOff		Invalidates the session on the server and clears the token.
Ping		Keeps the session alive. The server session expires after a period of inactivity; issue Ping periodically when no other REST call is being made.
GetServiceStatus		Reports the current service status (no authentication required).

Token Flow

The token returned by **LogOn** is attached to every subsequent REST request as the *UserName* HTTP header. The same token is reused by the streaming client as the *LS_password* field when the Lightstreamer session is created. There is no need to call LogOn twice: calling **Connect** on the main client performs the REST LogOn first and only then opens the streaming socket.

Sample

Open a REST session and call Ping

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey  = "<appkey>";
ShowMessage(oForex->REST_API->LogOn(oForex->Credentials->UserName,
                                    oForex->Credentials->Password,
                                    oForex->Credentials->AppKey));
ShowMessage(oForex->REST_API->Ping());
oForex->REST_API->LogOff();
```

Forex | Trade and Stop-Limit Orders

The REST API exposes five order-management endpoints. Each endpoint is provided in three overload forms so that the caller can choose the level of control required.

Endpoints

Method	Description
NewTradeOrder	Places a market / at-best trade order on a CFD market.
UpdateTradeOrder	Attaches or changes attached stop / limit protection on an existing open trade.
NewStopLimitOrder	Places a pending stop or limit order (not yet in the market).
UpdateStopLimitOrder	Amends an existing pending stop or limit order.
CancelOrder	Cancels a pending stop or limit order.

Overload Tiers

Order methods are available in two overloads:

1. **Scalar-parameter overload** (NewTradeOrder and CancelOrder only) — takes the most-used fields (MarketId, TradingAccountId, Direction, Quantity, BidPrice / OfferPrice, AuditId for NewTradeOrder; OrderId, TradingAccountId, MarketId for CancelOrder). The helper builds the request internally. Useful for plain, protection-less orders.
2. **Typed order-object overload** — takes a **TsgcHTTPForexTradeOrder**, **TsgcHTTPForexStopLimitOrder** or **TsgcHTTPForexCancelOrder**. These objects expose every documented field including **IfDone** (attached stop / limit brackets), **Close** arrays (close-by-order-id semantics), **PositionMethodId**, **Reference** and partial-fill flags. This is the recommended path for anything beyond a plain market order; the object-building convention is shared with other venues such as Kucoin.

TsgcHTTPForexTradeOrder Fields

- **MarketId:** numeric market id returned by ListCfdMarkets or FullSearchWithTags.
- **TradingAccountId:** the active trading account. Read it from the main client after LogOn.
- **Direction:** *buy* or *sell*.
- **Quantity:** trade size in the market's native unit.
- **BidPrice / OfferPrice:** prices observed by the client when the order was submitted (Forex.com uses these to detect stale quotes).
- **AuditId:** price audit id from the latest price tick.
- **PositionMethodId:** netting / hedging behaviour (*1 = Netting, 2 = Hedging*).
- **IfDone:** list of attached orders (stop, limit, guaranteed stop, trailing stop) that activate when the parent fills.
- **Close:** array of order ids to close when this order fills (partial close).
- **Reference:** caller-supplied client reference, echoed back in order updates.

Sample

Buy 1000 EUR/USD at market with an attached take-profit and stop-loss

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey  = "<appkey>";
oForex->Connect();

TsgcHTTPForexTradeOrder *oOrder = new TsgcHTTPForexTradeOrder();
oOrder->MarketId         = 401484830;
oOrder->TradingAccountId = oForex->TradingAccountId;
oOrder->Direction        = fdBuy;
oOrder->Quantity         = 1000;
oOrder->BidPrice         = 1.09120;
oOrder->OfferPrice       = 1.09130;
```

```
ShowMessage(oForex->NewTradeOrder(oOrder));
delete oOrder;
```

Forex | Account Queries

These endpoints read account state and order history. They require an active session (see Connect REST API).

Endpoints

Method	Parameters	Description
GetClientAndTradingAccount		Returns the client account and the list of trading accounts linked to the authenticated user.
ListOpenPositions	TradingAccountId	Returns all currently open positions on a trading account.
ListActiveStopLimitOrders	TradingAccountId	Returns all pending stop and limit orders.
GetOrder	OrderId	Returns a single order by id, including fill history and attached IfDone orders.
ListTradeHistory	TradingAccountId, MaxResults	Returns the trade-order history for the account, most recent first.
ListStopLimitOrderHistory	TradingAccountId, MaxResults	Returns the stop-limit order history for the account, most recent first.
SimulateTrade	JSON / scalar / order-object	Runs a trade order through the server validator without submitting it. Useful for pre-flight margin and price-tolerance checks. Accepts the same three overload tiers as NewTradeOrder.

Retired Endpoints

The official Forex.com / StoneX / CityIndex reference still documents a handful of endpoints that the live server no longer serves. These are intentionally not exposed by the component:

- **GetSystemLookup** — the reference describes a generic lookup endpoint, but the server responds with HTTP 404 for every known lookup key. Use **GetClientAndTradingAccount** and **ListCfdMarkets** instead.
- **/news/headlines/*** — the news-feed family is still listed in the reference but returns HTTP 500 on the live server. No replacement is available through the Trading API; news is served through the web front end only.

Sample

List open positions and the last 50 trades

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey  = "<appkey>";
oForex->Connect();
ShowMessage(oForex->REST_API->ListOpenPositions(oForex->TradingAccountId));
ShowMessage(oForex->REST_API->ListTradeHistory(oForex->TradingAccountId, 50));
```

Forex | Market Data

Static market metadata and historical prices. These endpoints are used to discover **MarketId** values that are then fed to the trade endpoints and to the streaming adapters.

Endpoints

Method	Parameters	Description
GetMarketInformation	MarketId	Returns the full market descriptor (symbol, decimal places, pip size, margin factor, trading hours).
ListCfdMarkets	SearchByMarketName, ClientAccountId, MaxResults	Lists CFD markets matching a name prefix.
FullSearchWithTags	Query, Tags, MaxResults	Free-text search across all markets with optional tag filters (FX, Indices, Shares, Commodities).
GetPriceBars	MarketId, Interval, Span, PriceBars	Historical OHLC bars. Interval is one of MINUTE, HOUR, DAY, WEEK; Span is the number of intervals per bar.
GetPriceTicks	MarketId, PriceTicks	Most-recent tick snapshot for a market (bid / ask / mid).

Sample

Look up EUR/USD and read the last 100 1-minute bars

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey  = "<appkey>";
oForex->Connect();
ShowMessage(oForex->REST_API->ListCfdMarkets("EUR/USD", 0, 10));
ShowMessage(oForex->REST_API->GetPriceBars(401484830, "MINUTE", 1, 100));
```

Forex | Connect WebSocket API

Forex.com / StoneX / CityIndex streams market data and account updates through Lightstreamer. The component speaks the native Lightstreamer TLCP 2.5 wire protocol over a WebSocket; no Lightstreamer native library is required.

Transport

- **WebSocket subprotocol:** *TLCP-2.5.0.lightstreamer.com*. Advertised in the *Sec-WebSocket-Protocol* request header.
- **Adapter set:** *STREAMINGALL*. Configured on the internal **Lightstreamer** subprotocol client; all Forex.com data adapters (PRICES, ORDERS, QUOTES, CLIENTACCOUNTMARGIN, TRADEMARGIN) live under this adapter set.
- **Authentication:** the session token obtained from the REST **LogOn** call is sent as *LS_password* when creating the Lightstreamer session, together with *LS_user* set to the account user name.

Reconnection

When **AutoReconnect** is enabled (the default) the underlying WebSocket client WatchDog reopens the socket after network errors. The Lightstreamer protocol itself signals *LOOP* frames when the server wants the client to move to a different URL or rebind the session; the component handles those transparently and only raises **OnForexDisconnect** / **OnForexConnect** around full session breaks.
When the REST session expires (no Ping for a long time) the streaming server closes the session. The component calls **LogOn** again using the stored credentials, refreshes **SessionToken**, and restarts the Lightstreamer session with the new token.

Sample

Open the streaming session

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey  = "<appkey>";
oForex->OnForexConnect    = ForexConnected;
oForex->OnForexDisconnect = ForexDisconnected;
oForex->OnForexException  = ForexException;
oForex->AutoReconnect     = true;
oForex->Connect();
oForex->Disconnect();
```

Forex | Subscribe to Streaming Adapters

Streaming data is delivered through five Lightstreamer data adapters under the *STREAMINGALL* adapter set. Each adapter has its own item naming convention and subscription mode — **MERGE** for snapshot-style state (current price, current account margin) and **RAW** for event-stream channels (order updates, position updates, quote events).

Adapters

DataAdapter	Item	Mode	Event	Fields
PRICES	ID.{MarketId}	MERGE	OnForexPriceTick	MarketId, TickDate, Bid, Offer Price, High, Low, Change, Direction, AuditId, StatusSummary
ORDERS	ORDERS.{ClientAccountId}	RAW	OnForexOrderUpdate	OrderId, MarketId, TradingAccountId, Direction, Quantity, Price, Status, OrderType, Currency, LastChangedDateTimeUTC
QUOTES	QUOTES.{ClientAccountId}	RAW	OnForexQuote	QuoteId, OrderId, RequestDateTimeUTC, ApprovalDateTimeUTC, BidPrice, OfferPrice Status
CLIENTACCOUNTMARGIN	ID.{ClientAccountId}	MERGE	OnForexAccountMargin	ClientAccountId, Cash, Margin MarginIndicator, NetEquity, OpenTradeEquity, TotalMarginRequirement, TradeableFunds, CurrencyId, CurrencyISOCode
TRADEMARGIN	TRADEMARGIN.{ClientAccountId}	RAW	OnForexPositionUpdate	OrderId, ClientAccountId, MaketId, MarketTypeId, DirectionId, Direction, Quantity, OpenPrice, CurrentPrice, Multiplier, MarginRequirement, Currency, OpenTradeEquity

Notes:

- PRICES and CLIENTACCOUNTMARGIN use the unified *ID.{numericId}* item form and stream a single mutable record per item (MERGE).
- ORDERS, QUOTES and TRADEMARGIN use the legacy per-adapter prefix (*ORDERS.{id}*, *QUOTES.{id}*, *TRADEMARGIN.{id}*) and stream a fire-hose of events (RAW). One item carries every order / quote / trade for the logged-in **ClientAccountId**; the records identify themselves by **OrderId** / **QuoteId**.
- The TRADEMARGIN record matches the official *TradeMarginDTO* from the legacy CityIndex SDK. **DirectionId** arrives as a numeric code (1 = buy, 2 = sell); the library also exposes **Direction** as the string *"buy"* / *"sell"*. **OpenPrice** is the trade entry price (server field *PriceTakenAt*), **CurrentPrice** the latest valuation price (*PriceCalculatedAt*), **OpenTradeEquity** the floating P&L in account currency (*OTEConverted*).
- The ORDERS **Direction** field is the numeric code published by the server (*1* / *2*); **Status** is a string (e.g. *"Accepted"*, *"Open"*, *"Cancelled"*); **OrderType** is a string (*"Market"* / *"Stop"* / *"Limit"*).
- The TRADEMARGIN feed does not publish per-trade **TradingAccountId**, **MarketName** or a per-trade timestamp; one item carries every position for the logged-in **ClientAccountId**. Call **REST_API.ListOpenPositions** for those fields. The ORDERS feed does not publish **TriggerPrice** nor **Applicability**; call **REST_API.GetOrder** when those are required.

Helper Methods

The main client wraps the Lightstreamer SUB / UNSUB plumbing. The common ones are:

- **WatchMarket(aMarketId):** subscribes to PRICES for one market. Updates arrive in **OnForexPriceTick**.
- **UnwatchMarket(aMarketId):** removes the subscription.
- **WatchAccount:** subscribes to ORDERS, CLIENTACCOUNTMARGIN and TRADEMARGIN for the logged-in accounts. Updates arrive in the matching events.
- **WatchQuotes:** subscribes to the QUOTES adapter for request-for-quote workflows.

Subscription Lifecycle Events

The main client raises two lifecycle events that mirror the underlying TLCP SUBOK / UNSUB frames:

- **OnForexSubscribe(Sender, aAdapter, aGroup, aItems, aFields):** fired when the server confirms any Lightstreamer subscription. *aAdapter* is the data adapter name (PRICES, ORDERS, QUOTES, CLIENTACCOUNTMARGIN, TRADEMARGIN); *aGroup* is the item group (e.g. *ID.<ClientAccountId>* or *ORDERS.<ClientAccountId>*); *aItems* and *aFields* are the counts reported by the server in the SUBOK / SUBCMD frame.
- **OnForexUnsubscribe(Sender, aAdapter, aGroup):** fired when a subscription ends — via **UnwatchMarket**, on streaming disconnect, or when the server sends an UNSUB frame.

Sample

Subscribe to EUR/USD price ticks

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey  = "<appkey>";
oForex->OnForexPriceTick = ForexPriceTick;
oForex->Connect();
oForex->WatchMarket(401484830);
```

API Blockchain

Blockchain

Blockchain WebSocket API allows developers to receive Real-Time notifications about new transactions and blocks.

Once WebSocket is open you can subscribe to a channel:

- **SubscribeTransactions:** Subscribe to notifications for all new bitcoin transactions.
- **UnsubscribeTransactions:** Unsubscribe from notifications for all new bitcoin transactions.

- **SubscribeAddress:** Receive new transactions for a specific bitcoin address.
- **UnSubscribeAddress:** Stop receiving new transactions for a specific bitcoin address.

Transactions are received **OnNewTransaction** Event:


```
{
    "op": "utx",
    "x": {
        "lock_time": 0,
        "ver": 1,
        "size": 192,
        "inputs": [
            {
                "sequence": 4294967295,
                "prev_out": {
                    "spent": true,
                    "tx_index": 99005468,
                    "type": 0,
                    "addr": "1BwGf3z7n2fHk6NoVJNkV32qwyAYsMhkWf",
                    "value": 65574000,
                    "n": 0,
                    "script": "76a91477f4c9ee75e449a74c21a4decfb50519cbc245b388ac"
                },
                "script": "483045022100e4ff962c292705f051c2c2fc519fa775a4d8955bce1a3e29884b2785277999ed02200b537e
            }
        ],
        "time": 1440086763,
        "tx_index": 99006637,
        "vin_sz": 1,
        "hash": "0857b9de1884eec314ecf67c040a2657b8e083e1f95e31d0b5ba3d328841fc7f",
        "vout_sz": 1,
        "relayed_by": "127.0.0.1",
        "out": [
            {
                "spent": false,
                "tx_index": 99006637,
                "type": 0,
                "addr": "1A828tTnkVFJfSvLCqF42ohZ51ksS3jJgX",
                "value": 65564000,
                "n": 0,
                "script": "76a914640cfdf7b79d94d1c980133e3587bd6053f091f388ac"
            }
        ]
    }
}
```


- **SubscribeBlocks:** Receive notifications when a new block is found. Note: if the chain splits you will receive more than one notification for a specific block height.
- **UnSubscribeBlocks:** Stop receiving notifications when a new block is found. Note: if the chain splits you will receive more than one notification for a specific block height.

Blocks are received **OnNewBlock** event:


```
{
    "op": "block",
    "x": {
```

```
        "txIndexes": [
            3187871,
            3187868
        ],
        "nTx": 0,
        "totalBTCSent": 0,
        "estimatedBTCSent": 0,
        "reward": 0,
        "size": 0,
        "blockIndex": 190460,
        "prevBlockIndex": 190457,
        "height": 170359,
        "hash": "00000000000006436073c07dfa188a8fa54fefadf571fd774863cda1b884b90f",
        "mrklRoot": "94e51495e0e8a0c3b78dac1220b2f35ceda8799b0a20cfa68601ed28126cfcc2",
        "version": 1,
        "time": 1331301261,
        "bits": 436942092,
        "nonce": 758889471
    }
}
```

API Cex

Cex

WebSocket API allows getting real-time notifications without sending extra requests, making it a faster way to obtain data from the exchange
Cex component has a property called Cex where you can fill API Keys provided by Cex to get access to your account data.

Message encoding

- All messages are encoded in JSON format.
- Prices are presented as strings to avoid rounding errors at JSON parsing on the client side.
- Compression of WebSocket frames is not supported by the server.
- Time is presented as integer UNIX timestamp in seconds.

Authentication

To get access to CEX.IO WebSocket data, you should be authorized.

- Log in to CEX.IO account.
- Go to https://cex.io/trade/profile#/api page.
- Select the type of required permissions.
- Click "Generate Key" button and save your secret key, as it will become inaccessible after activation.
- Activate your key.

Connectivity

- If a connected Socket is inactive for 15 seconds, CEX.IO server will send a PING message.
- Only server can be an Initiator of PING request.
- The server sends ping only to the authenticated user.
- The user has to respond with a PONG message. Otherwise, the WebSocket will be DISCONNECTED. This is handled automatically by the library.
- For the authenticated user, in case there is no notification or ping from the server within 15 seconds, it would be safer to send a request like 'ticker' or 'get-balance' and receive a response, in order to ensure connectivity and authentication.

Properties

- **Cex.Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false). The order book subscription is replayed too, but as it is a private channel it is held until the application authenticates again and the authentication is acknowledged.

Public Channels

These channels don't require authentication. Responses from the server are received through the OnCexMessage event.

- **SubscribeTickers:** Ticker feed with only price of transaction made on all pairs (deprecated)

```
{
    "e": "tick",
    "data": {
        "symbol1": "BTC",
        "symbol2": "USD",
        "price": "428.0123"
    }
}
```

- **SubscribeChart:** OHLCV chart feeds with Open, High, Low, Close, Volume numbers (deprecated)

```
{
    'e': 'ohlcv24',
    'pair': 'BTC:USD',
    'data': [
        '418.2936',
        '420.277',
        '412.09',
        '416.9778',
        '201451078368'
    ]
}
```

- **Subscribe Pair:** Market Depth feed (deprecated)

```
{
    'e': 'md_groupped',
    'data': {
        'pair': 'BTC:USD',
        'id': 11296131,
        'sell': {
            '427.5000': 1000000,
            '480.0000': 263544334,
            ...
        },
        'buy': {
            '385.0000': 3630000,
            '390.0000': 1452458642,
            ... 400+ pairs togather with 'sell' pairs
        }
    }
}
```

- **Subscribe Pair:** Order Book feed (deprecated)

```
{
    'e': 'md',
    'data': {
        'pair': 'BTC:USD',
        'buy_total': 63221099,
        'sell_total': 112430315118,
        'id': 11296131,
        'sell': [
            [426.45, 10000000],
            [426.5, 66088429300],
            [427, 1000000],
            ... 50 pairs overaall
        ],
        'buy': [
            [423.3, 4130702],
            [423.2701, 10641168],
            [423.2671, 1000000],
            ... 50 pairs overaall
        ]
    }
}
```

Private Channels

To access these channels, first call the Authenticate method. Responses from the server are received through the OnCexMessage event.

GetTicker


```
{
    "e": "ticker",
    "data": {
        "timestamp": "1471427037",
        "low": "290",
        "high": "290",
        "last": "290",
        "volume": "0.02062068",
        "volume30d": "14.38062068",
        "bid": 240,
        "ask": 290,
        "pair": [
            "BTC",
            "USD"
        ]
    },
    "oid": "1471427036908_1_ticker",
    "ok": "ok"
}
```


GetBalance


```
{
    "e": "get-balance",
    "data": {
           "balance": {
            'LTC': '10.00000000',
            'USD': '1024.00',
            'RUB': '35087.98',
            'EUR': '217.53',
            'GHS': '10.00000000',
            'BTC': '9.00000000'
        },
           "obalance": {
              'BTC': '0.12000000',
              'USD': "512.00",
           },
       },
       "time": 1435927928597
    "oid": "1435927928274_2_get-balance",
    "ok": "ok"
}
```


SubscribeOrderBook


```
{
"e": "order-book-subscibe",
"data": {
   "timestamp": 1435927929,
   "bids": [
      [
         241.947,
         155.91626
      ],
      [
         241,
         981.1255
      ],
   ],
   "asks": [
      [
         241.95,
         15.4613
      ],
      [
```

```
            241.99,
            17.3303
         ],
      ],
      "pair": "BTC:USD",
      "id": 67809
   },
   "oid": "1435927928274_5_order-book-subscribe",
   "ok": "ok"
   }
```

UnSubscribeOrderBook

```
{
"e": "order-book-unsubscribe",
"data": {
   "pair": "BTC:USD"
},
"oid": "1435927928274_4_order-book-unsubscribe",
"ok": "ok"
}
```

GetOpenOrders

```
{
"e": "open-orders",
"data": [
   {
      "id": "2477098",
      "time": "1435927928618",
      "type": "buy",
      "price": "241.9477",
      "amount": "0.02000000",
      "pending": "0.02000000"
   },
   {
      "id": "2477101",
      "time": "1435927928634",
      "type": "sell",
      "price": "241.9493",
      "amount": "0.02000000",
      "pending": "0.02000000"
   }
],
"oid": "1435927928274_9_open-orders",
"ok": "ok"
}
```

PlaceOrder

```
{
"e": "place-order",
"data": {
   "complete": false,
   "id": "2477098",
   "time": 1435927928618,
   "pending": "0.02000000",
   "amount": "0.02000000",
   "type": "buy",
   "price": "241.9477"
},
"oid": "1435927928274_7_place-order",
"ok": "ok"
}
```

CancelReplaceOrder

```
{
"e": "cancel-replace-order",
"data": {
   "complete": false,
   "id": "2689009",
   "time": 1443464955904,
   "pending": "0.04000000",
   "amount": "0.04000000",
   "type": "buy",
   "price": "243.25"
},
"oid": "1443464955209_16_cancel-replace-order",
"ok": "ok"
}
```


GetOrderRequest

In CEX.IO system, orders can be present in the trade engine or in an archive database. There can be time periods (~2 seconds or more), when the order is done/cancelled, but still not moved to the archive database. That means you cannot see it using calls: archived-orders/open-orders. This call allows getting order information in any case. Responses can have different format depending on orders location.


```
{
"e": "get-order",
"data": {
   "user": "XXX",
   "type": "buy",
   "symbol1": "BTC",
   "symbol2": "USD",
   "amount": "0.02000000",
   "remains": "0.02000000",
   "price": "50.75",
   "time": 1450214742160,
   "tradingFeeStrategy": "fixedFee",
   "tradingFeeBuy": "5",
   "tradingFeeSell": "5",
   "tradingFeeUserVolumeAmount": "nil",
   "a:USD:c": "1.08",
   "a:USD:s": "1.08",
   "a:USD:d": "0.00",
   "status": "a",
   "orderId": "5582060"
},
"oid": "1450214742135_10_get-order",
"ok": "ok"
}
```


CancelOrderRequest


```
{
"e": "cancel-order",
"data": {
   "order_id": "2477098"
   "time": 1443468122895
},
"oid": "1435927928274_12_cancel-order",
"ok": "ok"
}
```


GetArchivedOrders


```
{
"e": "archived-orders",
"data": [
   {
       "type": "buy",
       "symbol1": "BTC",
```

```
        "symbol2": "USD",
        "amount": 0,
        "amount2": 5000,
        "remains": 0,
        "time": "2015-04-17T10:46:27.971Z",
        "tradingFeeBuy": "2",
        "tradingFeeSell": "2",
        "ta:USD": "49.00",
        "fa:USD": "0.98",
        "orderId": "2340298",
        "status": "d",
        "a:BTC:cds": "0.18151851",
        "a:USD:cds": "50.00",
        "f:USD:cds": "0.98"
    },
    {
        "type": "buy",
        "symbol1": "BTC",
        "symbol2": "USD",
        "amount": 0,
        "amount2": 10000,
        "remains": 0,
        "time": "2015-04-08T15:46:04.651Z",
        "tradingFeeBuy": "2.99",
        "tradingFeeSell": "2.99",
        "ta:USD": "97.08",
        "fa:USD": "2.91",
        "orderId": "2265315",
        "status": "d",
        "a:BTC:cds": "0.39869578",
        "a:USD:cds": "100.00",
        "f:USD:cds": "2.91"
    }
],
"oid": "1435927928274    15_archived-orders",
"ok": "ok"
}
```


OpenPosition


```
{
    "e": "open-position",
    "oid": "1435927928274_7_open-position",
    "data": {
        'amount': '1',
        'symbol': 'BTC',
        "pair": [
            "BTC",
            "USD"
        ],
        'leverage': '2',
        'ptype': 'long',
        'anySlippage': 'true',
        'eoprice': '650.3232',
        'stopLossPrice': '600.3232'
    }
}
```


GetPosition


```
{
  "e": "get_position",
  "ok": "ok",
  "data": {
    "user": "ud100036721",
    "pair": "BTC:USD",
    "amount": "1.00000000",
    "symbol": "BTC",
    "msymbol": "USD",
    "omamount": "1528.77",
    "lsymbol": "USD",
    "lamount": "3057.53",
    "slamount": "3380.11",
    "leverage": "3",
    "stopLossPrice": "3380.1031",
    "dfl": "3380.10310000",
    "flPrice": "3057.53333333",
```

```
    "otime": 1513002370342,
    "psymbol": "BTC",
    "ptype": "long",
    "ofee": "10",
    "pfee": "10",
    "cfee": "10",
    "tfeeAmount": "152.88",
    "rinterval": "14400000",
    "okind": "Manual",
    "a:BTC:c": "1.00000000",
    "a:BTC:s": "1.00000000",
    "oorder": "89101551",
    "pamount": "1.00000000",
    "lremains": "3057.53",
    "slremains": "3380.11",
    "oprice": "4586.3000",
    "status": "a",
    "id": "125531",
    "a:USD:cds": "4739.18"
  }
}
```

GetOpenPositions

```
{
    'e': 'open_positions',
    "oid": "1435927928256_7_open-positions",
    'ok': 'ok',
    'data': [
        {
            'user': 'ud100036721',
            'id': '104102',
            'otime': 1475602208467,
            'symbol': 'BTC',
            'amount': '1.00000000',
            'leverage': '2',
            'ptype': 'long',
            'psymbol': 'BTC',
            'msymbol': 'USD',
            'lsymbol': 'USD',
            'pair': 'BTC:USD',
            'oprice': '607.5000'
            'stopLossPrice': '520.3232',
            'ofee': '1',
            'pfee': '3',
            'cfee': '4',
            'tfeeAmount': '3.04',
            'pamount': '1.00000000',
            'omamount': '303.75',
            'lamount': '303.75',
            'oorder': '34106774',
            'rinterval': '14400000',
            'dfl': '520.32320000',
            'slamount': '520.33',
            'slremains': '520.33',
            'lremains': '303.75',
            'flPrice': '303.75000000',
            'a:BTC:c': '1.00000000',
            'a:BTC:s': '1.00000000',
            'a:USD:cds': '610.54',
        },
        ...
    ]
}
```

ClosePosition

```
{
    'e': 'close_position',
    "oid": "1435927928364_7_close-position",
    'ok': 'ok',
    'data': {
        'id': 104034,
        'ctime': 1475484981063,
        'ptype': 'long',
        'msymbol': 'USD'
        'pair': {
```

```
            'symbol1': 'BTC',
            'symbol2': 'USD'
        }
        'price': '607.1700',
        'profit': '-12.48',
    }
}
```

API Cex Plus

Cex Plus

APIs supported

- WebSockets API: connect to a public websocket server and provides real-time market data updates.

Properties

- **CexPlus.Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

WebSockets API

WebSocket is a TCP-based full-duplex communication protocol. Full-duplex means that both parties can send each other messages asynchronously using the same communication channel. This section describes which messages should Exchange Plus and Client send each other. All messages should be valid JSON objects.

WebSocket API is mostly used to obtain information or do actions which are not available or not easy to do using REST API. However, some requests or actions are possible to do in both REST API and WebSocket API. Exchange Plus sends messages to Client as a response to request previously sent by Client, or as a notification about some event (without prior Client's request).

Public API Calls

Public API rate limit is implied in order to protect the system from DDoS attacks and ensuring all Clients can have same level of stable access to Exchange Plus API endpoints. Public requests are limited by IP address from which public API requests are made. Request limits are determined from cost associated with each public API call. By default, each public request has a cost of 1 point, but for some specific requests this cost can be higher. See up-to-date request rate limit cost information in specification of each method.

Exchange Plus limits Public API calls to maximum of 100 points per minute, considering that each Public API call has its cost (see below). If request rate limit is reached then Exchange Plus replies with error, sends disconnected event to Client and closes WS connection afterwards. Exchange Plus will continue to serve Client starting from the next calendar minute. In the following example, request counter will be reset at 11:02:00.000.

Method	Description
GetTicker	This method is designed to obtain current information about Ticker, including data about current prices, 24h price & volume changes, last trade event etc. of certain assets.
GetOrderBook	This method allows Client to receive current order book snapshot for specific trading pair.
GetCandles	By using Candles method Client can receive historical OHLCV candles of different resolutions and data types. Client can indicate additional timeframe and limit filters to make response more precise to Client's requirements.
GetTradeHistory	This method allows Client to obtain historical data as to occurred trades upon requested trading pair. Client can supplement Trade History request with additional filter parameters, such as timeframe period, tradeIds range, side etc. to receive trades which match request parameters.
GetServerTime	This method is used to get the current time on Exchange Plus server. It can be useful for applications that have to be synchronized with the server's time.

GetPairsInfo	Pair Info method allows Client to receive the parameters for all supported trading pairs.
GetCurrenciesInfo	Currencies Info method allows Client to receive the parameters for all currencies configured in Exchange Plus as well as the deposit and withdrawal availability between Exchange Plus and CEX.IO Wallet.
GetProcessingInfo	This request allows Client to receive detailed information about available options to make deposits from external wallets and withdrawals to external wallets as to each supported cryptocurrency, including cryptocurrency name and available blockchains for deposit\withdrawals. Also, as to each supported blockchain there are indicated type of cryptocurrency on indicated blockchain, current deposit\withdrawal availability, minimum amounts for deposits\withdrawals, external withdrawal fees. Processing Information makes Client more flexible in choosing desired blockchain for receiving Deposit address and initiating external withdrawals via certain blockchain, so that Client uses more convenient way of transferring his crypto assets to or from CEX.IO Ecosystem.
SubscribeOrderBook	Client by subscribing via WebSocket can subscribe to order book feed upon requested trading pair. In response to Order Book Subscribe request Client will receive current (initial) order book snapshot for requested pair with indicated seqId number. To track following updates to Order Book Client needs to subscribe via WebSocket to “order_book_increment“ messages, which would contain trading pair name, seqId number, Bids and Asks price levels deltas.
UnSubscribeOrderBook	UnSubscribe from the order book channel.
SubscribeTrade	By using the Trade Subscribe method Client can subscribe via WebSocket to live feed of trade events which occur on requested trading pair. In response to Trade Subscribe request Client will receive a unique identifier of trade subscription which should further be used for unsubscription when trade subscription is not longer needed for Client. Client should subscribe via WebSocket to “tradeHistorySnapshot” and “tradeUpdate” messages to receive initial and periodical Trade History snapshots, and live trade events for requested trading pair.
UnSubscribeTrade	UnSubscribe from the trade channel.

Example: get the latest ticker of BTC-USD pair

```
TsgcWebSocketClient* oClient;
TsgcWSAPI_CexPlus* oCexPlus;

void __fastcall OnCexPlusConnectEvent(System::TObject* Sender)
{
    oCexPlus->GetTicker("BTC-USD");
}

void __fastcall OnCexPlusMessageEvent(System::TObject* Sender, UnicodeString Event, UnicodeString Msg)
{
    ShowMessage("Ticker data: " + Msg);
}

void SetupCexPlus()
{
    oClient = new TsgcWebSocketClient(NULL);
    oCexPlus = new TsgcWSAPI_CexPlus(NULL);
    oClient->Active = true;
    oCexPlus->Client = oClient;
    oCexPlus->OnConnect = OnCexPlusConnectEvent;
    oCexPlus->OnMessage = OnCexPlusMessageEvent;
}
```

Private API Calls

Exchange Plus uses API keys to allow access to Private APIs.

Client can generate, configure and manage api keys, set permission levels, whitelisted IPs for API key etc. via Exchange Plus Web Terminal in the API Keys Management Profile section.

API Keys limit: By default Client can have up to 5 API Keys.

To restrict access to certain functionality while using of API Keys there should be defined specific set of permissions for each API Key. The defined set of permissions can be edited further if necessary.

The following permission levels are available for API Keys:

- **Read:** permission level for viewing of account related data, receiving reports, subscribing to market data etc.
- **Trade:** permission level, which allows placing and cancelling orders on behalf of account.
- **Funds Internal:** permission level, which allows transferring funds between accounts (between sub-accounts or main account and sub-accounts) of CEX.IO Exchange Plus Portfolio.
- **Funds Wallet:** permission level, which allows transferring funds from CEX.IO Exchange Plus Portfolio accounts (main account and sub-accounts) to CEX.IO Wallet and vice versa.

Method	Description
GetCurrentFee	This method indicates current fees at specific moment of time with consideration of Client' up-to-date 30d volume and day of week (fees can be different for e.g. on weekends).
GetFeeStrategy	Fee Strategy returns all fee options, which could be applied for Client, considering Client's trading volume, day of week, pairs, group of pairs etc. This method provides information about general fee strategy, which includes all possible trading fee values that can be applied for Client. To receive current trading fees, based on Client's current 30d trading volume, Client should use [Current Fee] method. To receive current 30d trading volume, Client should use [Volume] method.
GetVolume	This request allows Client to receive his trading volume for the last 30 days in USD equivalent.
CreateAccount	This request allows Client to create new sub-account. By default Client can have up to 5 sub-accounts, including main account.
GetAccountStatus	By using Account Status V3 method, Client can find out current balance and it's indicative equivalent in converted currency (by default "USD"), amounts locked in open (active) orders as to each sub-account and currency. If trading fee balance is available for Client, then response will also contain general trading fee balance data such as promo name, currency name, total balance and expiration date of this promo on Trading Fee Balance. It's Client's responsibility to track his sub-accounts available trading balance as current sub-account balance reduced by the balance amount locked in open (active) orders on sub-account.
GetOrders	This request allows Client to find out info about his orders.
NewOrder	Client can place new orders via WebSocket API by using Do My New Order Request. Along with a response to this request, Exchange Plus sends Account Event and Execution Report messages to Client if the request is successful. Response message indicates the last up-to-date status of order which is available in the system at the moment of sending the response. If the Client did not receive a Response message to Do My New Order Request - the Client can query current status of the order by using Get My Orders Request with clientOrderId parameter. When sending a request for new order, it is highly recommended to use clientOrderId parameter which corresponds to the specific new order request on the client's side. Exchange Plus avoids multiple placing the orders with the same clientOrderId. If more than one new orders with identical clientOrderId and other order parameters are identified - Exchange Plus places only the first order and returns the status of such order to the Client in response to the second and subsequent new order requests with the same parameters. If more than one new orders with identical clientOrderId but with different other order parameters are identified - Exchange Plus processes only the first order and rejects the second and subsequent new order requests with the same clientOrderID but with different other order parameters.

NewMarketOrder	Places a new market order. An overload accepts the amount of the order, in currency1 (**AmountCcy1**) and optionally in currency2 (**AmountCcy2**), only the amounts which are set are sent.
NewLimitOrder	Places a new limit order. An overload accepts the amount of the order, in currency1 (**AmountCcy1**) and optionally in currency2 (**AmountCcy2**), only the amounts which are set are sent.
CancelOrder	Client can cancel orders. Along with a response to this request, Exchange Plus sends Account Event and Execution Report messages to Client if this request is successful. Also, if request to cancel an order is declined, Exchange Plus sends Order Cancellation Rejection message.
CancelAllOrders	Client can cancel all open orders via WebSocket API. Along with a response to this request Exchange Plus will start cancellation process for all open orders and send corresponding Account Event and Execution Report messages to the Client.
GetTransactionHistory	This request allows Client to find out his financial transactions (deposits, withdrawals, internal transfers, commissions or trades).
GetFundingHistory	This request allows Client to find his deposit and withdrawal transactions.
InternalTransfer	Client can request to transfer money between his sub-accounts or between his main account and sub-account. Exchange Plus does not charge Client any commission for transferring funds between his accounts. Along with a response to this request, Exchange Plus sends Account Event messages to Client if this request is successful.
GetDepositAddress	This method can be used by Client for receiving a crypto address to deposit cryptocurrency. Deposit address can be generated for main and sub-accounts. The list of available blockchains for generating deposit address can be received by Client via Get Processing Info request.
FundsDepositFromWallet	Client can deposit funds from CEX.IO Wallet to Exchange Plus account. The system avoids processing of multiple deposit requests with the same clientTxId. If multiple deposit requests with identical clientTxId are received - the system processes only the first deposit request and rejects the second and subsequent deposit requests with the same clientTxId.
FundsWithdrawalToWallet	Client can withdraw funds from Exchange Plus account to CEX.IO Wallet. The system avoids multiple withdrawal requests with the same clientTxId. If multiple withdrawal requests with identical clientTxId are received - the system processes only the first withdrawal request and rejects the second and subsequent withdrawal requests with the same clientTxId.
GetWalletBalance	Retrieves the CEX.IO Wallet balance information.
SubscribeAccountEvents	Subscribes to real-time account event notifications (balance changes, order fills).
UnSubscribeAccountEvents	Unsubscribes from account event notifications.

Example: get the orders.

```
TsgcWebSocketClient* oClient;
TsgcWSAPI_CexPlus* oCexPlus;

void __fastcall OnCexPlusAuthenticatedEvent(System::TObject* Sender)
{
    oCexPlus->GetOrders();
}

void __fastcall OnCexPlusMessageEvent(System::TObject* Sender, UnicodeString Event, UnicodeString Msg)
{
    ShowMessage("Orders: " + Msg);
}

void SetupCexPlus()
{
```

```
    oClient = new TsgcWebSocketClient(NULL);
    oCexPlus = new TsgcWSAPI_CexPlus(NULL);
    oClient->Active = true;
    oCexPlus->Client = oClient;
    oCexPlus->CexPlus->ApiKey = "your-api-key";
    oCexPlus->CexPlus->ApiSecret = "your-api-secret";
    oCexPlus->OnCexPlusAuthenticated = OnCexPlusAuthenticatedEvent;
    oCexPlus->OnMessage = OnCexPlusMessageEvent;
}
```

API Discord

Discord
Note: As of API v10, the Discord API domain has changed from `discordapp.com` to `discord.com`. The component has been updated to use API version 10 and the new domain.

Gateways are Discord's form of real-time communication over secure WebSockets. Clients will receive events and data over the gateway they are connected to and send data over the REST API.

Authorization

First you must generate a new Bot, and copy Bot Token which will be used to authenticate through API. Then set this token in API Component.

```
TsgcWSAPI_Discord1->DiscordOptions->BotOptions->Token = "...bot token here...";
```

Intents

Maintaining a stateful application can be difficult when it comes to the amount of data you're expected to process, especially at scale. Gateway Intents are a system to help you lower that computational burden.
When identifying to the gateway, you can specify an intents parameter which allows you to conditionally subscribe to pre-defined "intents", groups of events defined by Discord. If you do not specify a certain intent, you will not receive any of the gateway events that are batched into that group. The valid intents are (zero value means all events are received):

GUILDS (1 << 0) = Integer (1)
 - GUILD_CREATE
 - GUILD_DELETE
 - GUILD_ROLE_CREATE
 - GUILD_ROLE_UPDATE
 - GUILD_ROLE_DELETE
 - CHANNEL_CREATE
 - CHANNEL_UPDATE
 - CHANNEL_DELETE
 - CHANNEL_PINS_UPDATE
GUILD_MEMBERS (1 << 1) = Integer (2)
 - GUILD_MEMBER_ADD
 - GUILD_MEMBER_UPDATE
 - GUILD_MEMBER_REMOVE
GUILD_BANS (1 << 2) = Integer (4)
 - GUILD_BAN_ADD
 - GUILD_BAN_REMOVE
GUILD_EMOJIS (1 << 3) = Integer (8)
 - GUILD_EMOJIS_UPDATE
GUILD_INTEGRATIONS (1 << 4) = Integer (16)
 - GUILD_INTEGRATIONS_UPDATE
GUILD_WEBHOOKS (1 << 5) = Integer (32)
 - WEBHOOKS_UPDATE
GUILD_INVITES (1 << 6) = Integer (64)
 - INVITE_CREATE
 - INVITE_DELETE
GUILD_VOICE_STATES (1 << 7) = Integer (128)
 - VOICE_STATE_UPDATE
GUILD_PRESENCES (1 << 8) = Integer (256)
 - PRESENCE_UPDATE

GUILD_MESSAGES (1 << 9) = Integer (512)
 - MESSAGE_CREATE
 - MESSAGE_UPDATE
 - MESSAGE_DELETE
GUILD_MESSAGE_REACTIONS (1 << 10) = Integer (1024)
 - MESSAGE_REACTION_ADD
 - MESSAGE_REACTION_REMOVE
 - MESSAGE_REACTION_REMOVE_ALL
 - MESSAGE_REACTION_REMOVE_EMOJI
GUILD_MESSAGE_TYPING (1 << 11) = Integer (2048)
 - TYPING_START
DIRECT_MESSAGES (1 << 12) = Integer (4096)
 - CHANNEL_CREATE
 - MESSAGE_CREATE
 - MESSAGE_UPDATE
 - MESSAGE_DELETE
 - CHANNEL_PINS_UPDATE
DIRECT_MESSAGE_REACTIONS (1 << 13) = Integer (8192)
 - MESSAGE_REACTION_ADD
 - MESSAGE_REACTION_REMOVE
 - MESSAGE_REACTION_REMOVE_ALL
 - MESSAGE_REACTION_REMOVE_EMOJI
DIRECT_MESSAGE_TYPING (1 << 14) = Integer (16384)
 - TYPING_START

Heartbeat

Heartbeats are automatically handled by the component so you don't need to worry about them. When the client connects to the server, the server sends a HELLO response with a heartbeat interval, and the component reads the response and automatically adjusts the heartbeat to send a ping every x seconds. Sometimes the server can send a ping to the client; this is handled automatically by the client too.

Connection Ready

When the connection is ready, after a successful login and authorization by the server, **OnDiscordReady** event is raised and then you can start to receive updates from server.

Connection Resume

If the connection closes unexpectedly, when the client tries to reconnect, it calls **OnDiscordBeforeReconnect** event. The component automatically saves all data needed to make a successful resume, but parameters can be changed if needed. If you don't want to reconnect and start a new clean session, just set Reconnect to False.

If session is resumed, **OnDiscordResumed** event is fired. If it's a new session, **OnDiscordReady** will be fired.

Dispatch Events

Events are dispatched through **OnDiscordDispatch**, so here you can read events sent by the server to the client.

```
void OnDiscordDispatch(TObject *Sender, const string aEvent, const string RawData)
{
  DoLog("#discord dispatch: " + aEvent + " " + RawData);
}
```

aEvent parameter contains the event name.
RawData contains full JSON message.

HTTP Requests

In order to request info about guilds, users, or update data... instead of using gateway websocket messages, Discord requires the use of HTTP requests, Find below all methods available to make an HTTP request:

function GET_Request(const aPath: String): string;
function POST_Request(const aPath, aMessage: String): string;
function PUT_Request(const aPath, aMessage: String): string;
function PATCH_Request(const aPath, aMessage: String): string;
function DELETE_Request(const aPath: String): string;

Example: get current user info

```
result = GET_Request("/users/@me");
```

sample response from server:

```
{
"id": "637423922035480852",
"username": "test",
"avatar": null,
"discriminator": "5125",
"bot": true,
"email": null,
"verified": true,
"locale": "en-US",
"mfa_enabled": false,
"flags": 0
}
```

API | OpenAI

The OpenAI Realtime API enables low-latency, multimodal interactions including speech-to-speech conversational experiences and real-time transcription.

The component **TsgcWSAPI_OpenAI** implements the RealTime OpenAI API.

Configuration

Use the **method** property to select **Conversation** or **Transcription**, currently only Transcription mode is supported.

The **InputAudio** property allows you to customize the following data:

- **Language:** example english (value = 'en').
- **Model:** which model will be used, example: whisper-1
- **Prompt:** optional prompt to provide instructions to the model.

OpenAI

The OpenAI API uses API keys for authentication. Visit your API Keys page to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code (browsers, apps). Production requests must be routed through your own backend server where your API key can be securely loaded from an environment variable or key management service.

This **API Key** must be configured in the **OpenAIOptions.ApiKey** property of the component. Optionally, for users who belong to multiple organizations, you can set your Organization in the property **OpenAIOptions.Organization** if your account belongs to an organization.

Once the API Key is configured, find below a list of available functions to interact with the OpenAI API.

Azure

The client supports Microsoft Azure OpenAI Services, so you can use your Azure account to interact with the Azure OpenAI API too. In order to configure the client to work with Azure, follow the next steps:

1. Configure the property **OpenAIOptions.Provider** = sgcoaipAzure
2. Set the values of ResourceName, DeploymentId and APIVersion (these values can be located in your Azure Account)
    1. **OpenAIOptions.AzureOptions.ResourceName** = <your resource name>.
    2. **OpenAIOptions.AzureOptions.DeploymentId** = <your deployment id>.
    3. **OpenAIOptions.AzureOptions.APIVersion** = <api version>, for example 2024-10-01-preview.
3. Set the API Key of your Azure Account
    1. **OpenAIOptions.ApiKey** = <azure api key>.

With the Azure provider both Methods connect to your own Azure resource, `wss://<ResourceName>.openai.azure.com/openai/realtime?api-version=<APIVersion>&deployment=<DeploymentId>`, so the API Key is sent to that host and to no other.

Transcription Example

Find below an example of real-time transcription using openAI API

```
TsgcWebSocketClient *WSClient = new TsgcWebSocketClient(NULL);
TsgcAudioRecorderWave *oAudio = new TsgcAudioRecorderWave(NULL);
TsgcWSAPI_OpenAI *OpenAI = new TsgcWSAPI_OpenAI(NULL);
OpenAI->Client = WSClient;
```

```
OpenAI->AudioRecorder = oAudio;

OpenAI->OpenAIOptions->APIKey = "your-api-key-here";
OpenAI->OpenAIOptions->Method = sgcoaimTranscription;
OpenAI->OpenAIOptions->Provider = sgcoaipOpenAI;
OpenAI->InputAudio->Language = "en";
OpenAI->InputAudio->Model = "whisper-1";

void __fastcall TForm1::OnOpenAIAudioTranscriptionCompleted(TObject *Sender, TsgcWSOpenAIConversation_Item_Comple
    Log("#transcription_completed: " + aItem->Transcript);
}
```

API MEXC

MEXC is a global cryptocurrency exchange that exposes streaming market and account data over secure WebSocket connections.

The WebSocket Spot API follows the specifications published in the Market Streams and User Data Streams documentation. Market channels are delivered as Protocol Buffers frames while private streams use JSON. The **TsgcWSAPI_MEXC** component manages the connection lifecycle, authentication and listen key maintenance so that applications can focus on processing the pushed data.

In addition to streaming over WebSockets, the toolkit also exposes the complete Spot HTTP API. The **TsgcHTTP_API_MEXC_Spot** component (unit **sgcHTTP_API_MEXC**) wraps every official REST endpoint so applications can query market data or trade with signed requests.

Component configuration

The base WebSocket endpoint is **wss://wbs-api.mexc.com/ws**. When **MEXCUserDataStreams.UserStream** is enabled and a valid API key is provided, the component automatically requests a listen key and appends it to the connection URL.

Drop a **TsgcWebSocketClient** and a **TsgcWSAPI_MEXC** component on the form (or create them in code), assign the client instance to the API component and activate the client to start the session.

```
TsgcWebSocketClient *WSClient = new TsgcWebSocketClient();
TsgcWSAPI_MEXC *MEXC = new TsgcWSAPI_MEXC();
MEXC->Client = WSClient;
MEXC->MEXCAPI->ApiKey = "YOUR_KEY";
MEXC->MEXCAPI->ApiSecret = "YOUR_SECRET";
MEXC->MEXCUserDataStreams->UserStream = true;
if (WSClient->Connect()) {
  MEXC->SubscribeTrade("BTCUSDT");
}
```

Properties

- **Client:** reference to the **TsgcWebSocketClient** transport. The component enables the client heartbeat (15 seconds) to keep the session alive.
- **MEXCAPI.ApiKey / ApiSecret:** Spot API credentials. The key is required for private channels; the secret is needed to sign listen key requests.
- **MEXCUserDataStreams.UserStream:** when **True** (default) the component retrieves a listen key before connecting and refreshes it every 10 minutes. Disable it to work with public feeds only.
- **MEXCUserDataStreams.ListenKeyOnDisconnect:** action executed when the WebSocket disconnects while a listen key is active:
  - **blkmxodDeleteListenKey:** revoke the listen key via REST (default).
  - **blkmxodClearListenKey:** keep the listen key server-side and clear the cached value.
  - **blkmxodDoNothing:** reuse the same listen key on the next connection.
- **ListenKey:** read-only property exposing the active listen key returned by the REST API.
- **RawMessages:** if enabled, incoming text frames are not parsed and are forwarded as plain JSON through the inherited WebSocket events.
- **MEXCAPI.Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Events

- **OnConnect / OnDisconnect:** raised when the underlying WebSocket session is established or terminated.

- **OnMEXCMarketStream:** triggered for every Protobuf market frame. The handler receives a **TsgcMEXCSpotProtoMessage** whose **MessageType** property identifies the payload (Trade, KLine, DiffDepth, LimitDepth, BookTicker, BookTickerBatch, MiniTicker or MiniTickers). Use the helper classes in **sgcWebSocket_API_MEXC_Proto** to inspect the decoded structure.
- **OnMEXCResponse:** receives subscription acknowledgements and ping/pong responses in JSON format (**Id**, **Code**, **Message**).
- **OnMEXCUserDataStream:** fired when private account messages arrive (orders, deals, balance updates). The channel name and raw JSON payload are supplied.
- **OnMEXCException:** captures REST or WebSocket exceptions (listen key errors, HTTP failures, parsing issues, etc.).

The following sample demonstrates how to decode a trade frame:

WebSocket Subscription methods

The component exposes helpers for each public Spot channel. Invoke the **Subscribe*** method to start streaming data and the corresponding **UnSubscribe*** to stop it.

Public channels

Method	Parameters	Description
SubscribeTrade	Symbol, Interval (ms)	Aggregated trade executions delivered as Protobuf **publicdeals**.
SubscribeKline	Symbol, Interval	Candlestick updates for configurable intervals (1 minute to 1 month).
SubscribeDiffDepth	Symbol, Interval (ms)	Order book incremental depth deltas for local book maintenance.
SubscribeBookDepth	Symbol, Levels	Periodic snapshots of the limit order book with the requested depth (default 5).
SubscribeBookTicker	Symbol, Interval (ms)	Best bid/ask updates for a single trading pair.
SubscribeBookTicker-Batch	UTC flag	Aggregated book ticker updates for multiple symbols.
SubscribeMiniTickers	UTC flag	Rolling 24h mini ticker statistics for all active instruments.
SubscribeMiniTicker	Symbol, UTC flag	Mini ticker for a single symbol.

Private channels

Private topics require a valid API key. The component subscribes using the active listen key and renews it periodically.

Method	Description
SubscribeAccountUpdate	Account balance and position changes.
SubscribeAccountDeals	Execution reports for filled orders.
SubscribeAccountOrders	Order life-cycle updates (new, cancelled, rejected, etc.).

Use the corresponding **UnSubscribe*** methods to terminate a stream. The component automatically sends **PING** commands and validates the returned **PONG** to monitor connectivity.

Public REST endpoints

Publicly available REST methods.

Method	Description
Ping	Connectivity check that calls **/api/v3/ping** and returns **True** when the exchange replies with an empty object.
GetServerTime	Retrieves the current exchange timestamp from **/api/v3/time**.
GetExchangeInformation	Returns trading rules and symbol metadata from **/api/v3/exchangeInfo**.
GetOrderBook	Downloads order book snapshots from **/api/v3/depth**; set the optional **limit** parameter to control the number of levels (default 100).
GetTrades	Retrieves recent public trades from **/api/v3/trades** with an optional **limit** (default 100).
GetHistoricalTrades	Provides historical trade data with optional **limit** and **fromId** filters.
GetAggregateTrades	Returns compressed/aggregated trades via **/api/v3/aggTrades** supporting optional **limit**, **fromId**, **startTime** and **endTime** filters.
GetKlines	Downloads candlesticks from **/api/v3/klines** supporting time range and limit filters.
GetAveragePrice	Returns the current weighted average price from **/api/v3/avgPrice**.
Get24hrTicker	Retrieves 24 hour ticker statistics for one or all symbols via **/api/v3/ticker/24hr**.
GetPriceTicker	Fetches the latest price using **/api/v3/ticker/price**; pass an empty symbol to retrieve prices for every trading pair.
GetBookTicker	Obtains best bid/ask quotes from **/api/v3/ticker/bookTicker** with support for single-symbol or full-exchange requests.

Private REST endpoints

Private endpoints require valid API keys and a signed query string. The component automatically appends the timestamp, recvWindow and HMAC signature.

Method	Description
GetAccountInformation	Returns balances and permissions from **/api/v3/account**.
GetOpenOrders	Lists current open orders (optionally filtered by symbol).
GetAllOrders	Retrieves historical orders with optional time and limit filters.
GetOrder	Queries the status of a specific order by supplying either **orderId** or **origClientOrderId**.
GetMyTrades	Lists private trade executions with optional **limit**, **fromId**, **startTime** and **endTime** filters.
NewOrder	Places a live order on **/api/v3/order** (market, limit, stop, etc.) with optional **timeInForce**, **quantity**, **quoteOrderQty**, **price**, **newClientOrderId**, **stopPrice**, **icebergQty** and extra parameters.
TestNewOrder	Sends a validation-only request to **/api/v3/order/test** accepting the same parameter set as **NewOrder**.
CancelOrder	Cancels a specific order on **/api/v3/order** using **orderId** or **origClientOrderId**.
CancelAllOrders	Bulk cancels all open orders for a symbol via **/api/v3/openOrders**.
GetSubAccounts	Lists managed sub-accounts (**/api/v3/sub-account/list**).
GetSubAccountAssets	Returns balances for a specific sub-account.

TransferSubAccount	Transfers assets between sub-accounts through **/api/v3/sub-account/transfer**; provide the amount and optionally the transfer **type**.
GetDepositAddress	Requests deposit addresses with an optional **network** filter.
GetDepositHistory	Fetches deposit records with optional **coin**, **status**, **startTime** and **endTime** filters.
GetWithdrawHistory	Returns withdrawal history filtered by **coin**, **status** and optional time range.
Withdraw	Submits a withdrawal request via **/api/v3/capital/withdraw/apply** including the mandatory **coin**, **address** and **amount** plus optional **network**, **addressTag**, **withdrawOrderId** and extra parameters.
BatchOrders	Submit batch orders for multiple symbols in a single request.
GetTradeFee	Get the trade fee rate for a specific symbol.
GetDepositHistory	Fetches deposit records with optional **coin**, **status**, **startTime** and **endTime** filters.
CancelWithdraw	Cancel a pending withdrawal request by withdrawal ID.
CreateInternalTransfer	Transfer assets between accounts internally.
GetTransferHistory	Get the history of internal transfers between accounts.

API MEXC Futures

MEXC perpetual and delivery contracts expose a dedicated streaming API documented in the official Futures WebSocket specification. The **TsgcWSAPI_MEXC_Futures** component encapsulates the connection, login handshake and topic management required to consume real-time derivatives data.

MEXC also maintains a REST interface for derivatives trading. The **TsgcHTTP_API_MEXC_Futures** component contained in **sgcHTTP_API_MEXC** complements the WebSocket feeds with HTTP helpers that map one-to-one to the official endpoints.

Component configuration

The Futures WebSocket endpoint is **wss://contract.mexc.com/edge**. Messages are encoded as JSON objects and the exchange requires periodic **ping**/**pong** frames which are handled automatically by the component.

To receive private notifications (order and account data) you must authenticate. Set **MEXCAPI.ApiKey** and **MEXCAPI.ApiSecret** before activating the WebSocket client. When credentials are present the component signs a login request (HMAC SHA256) as soon as the socket connects.

```
TsgcWebSocketClient *WSClient = new TsgcWebSocketClient();
TsgcWSAPI_MEXC_Futures *Futures = new TsgcWSAPI_MEXC_Futures();
Futures->Client = WSClient;
Futures->MEXCAPI->ApiKey = "YOUR_KEY";
Futures->MEXCAPI->ApiSecret = "YOUR_SECRET";
if (WSClient->Connect()) {
  Futures->SubscribeDepth("BTC_USDT", true);
}
```

Properties

- **Client:** WebSocket transport used by the API component. Heartbeat support is enabled to keep the socket active.
- **MEXCAPI.ApiKey / ApiSecret:** required to sign the login request. Without credentials only public channels are accessible.
- **RawMessages:** bypasses the JSON parser when enabled so that the raw text frames are exposed through the base events.
- **MEXCAPI.Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Events

- **OnConnect / OnDisconnect:** WebSocket lifecycle notifications.
- **OnMEXCLogin:** raised after a successful authenticated login.
- **OnMEXCSubscribed / OnMEXCUnsubscribed:** confirmation of subscription changes. The event exposes the channel name returned by MEXC.
- **OnMEXCMessage:** delivers every market or account update together with the channel identifier.
- **OnMEXCError:** triggered when the server replies with an **rs.error** payload (invalid parameters, authentication failure, etc.).
- **OnMEXCException:** reports local exceptions raised while processing JSON data or performing network operations.

The following handler records subscription confirmations and prints incoming depth snapshots:

WebSocket Subscription methods

Each helper wraps a **sub**/**unsub** request as described by the official API. Use the matching **UnSubscribe*** method to cancel the stream.

Method	Parameters	Description
SubscribeDeal	Symbol	Trades executed on the contract (channel: **deal**).
SubscribeTickers	–	Global ticker statistics for all contracts.
SubscribeTicker	Symbol	Ticker summary for a single instrument.
SubscribeDepth	Symbol, Compress	Incremental order book updates
SubscribeDepthFull	Symbol, Level	Full depth snapshots with configurable number of levels (default 20).
SubscribeKline	Symbol, Interval	Candlestick data for supported timeframes (Min1, Min5, Min15, Min30, Min60, Hour4, Hour12, Day1, Week1, Month1).
SubscribeFundingRate	Symbol	Latest funding rate announcements.
SubscribeIndexPrice	Symbol	Underlying index price stream.
SubscribeFairPrice	Symbol	Mark (fair) price updates pushed by the exchange.

When the server acknowledges a request the **OnMEXCSubscribed** event fires with the channel name (for example **rs.sub.depth.BTC_USDT**). Errors are forwarded through **OnMEXCError** including the server message for troubleshooting.

Private channels

Private channels require a valid API key. The component authenticates automatically when credentials are provided.

Method	Description
SubscribePersonalOrder	Personal order updates.
SubscribePersonalOrderDeal	Personal order deal (fill) updates.
SubscribePersonalPosition	Personal position updates.
SubscribePersonalPlanOrder	Personal plan (trigger) order updates.
SubscribePersonalStopOrder	Personal stop order updates.
SubscribePersonalStop-PlanOrder	Personal stop plan order updates.
SubscribePersonalRiskLimit	Personal risk limit updates.
SubscribePersonalADLLevel	Personal ADL (auto-deleveraging) level updates.
SubscribePersonalAsset	Personal asset updates.

Public REST endpoints

Publicly available REST methods.

Method	Description
GetPing	Checks API reachability via **/api/v1/ping**.
GetServerTime	Returns the server timestamp from **/api/v1/time**.

GetContracts	Provides the list of available contracts (**/api/v1/contract/detail**).
GetDepth	Downloads order book depth from **/api/v1/contract/depth**; use the optional **limit** parameter (default 50) to select the number of levels.
GetDeals	Retrieves recent trades using **/api/v1/contract/deals** with an optional **limit** (default 100).
GetKlines	Returns candlestick data through **/api/v1/contract/kline** with optional **startTime**, **endTime** and **limit** filters (default 200).
GetIndexPrice	Fetches the underlying index price for the requested symbol (**/api/v1/contract/index_price**).
GetFairPrice	Returns the fair (mark) price for a contract from **/api/v1/contract/fair_price**.
GetFundingRate	Reports the latest funding rate for a symbol using **/api/v1/contract/funding_rate**.
GetAllTickers	Get all futures contract tickers with real-time price summaries.
GetFundingRateHistory	Get historical funding rate records for a contract.
GetFairPriceKline	Get fair (mark) price kline/candlestick data for a contract.
GetIndexPriceKline	Get index price kline/candlestick data for a contract.

Private REST endpoints

Private derivatives endpoints require API credentials. The component signs each request with the timestamp and recvWindow automatically.

Breaking change: the signatures of **PlaceOrder** and **SetPositionLeverage** changed to match the contract endpoints published by MEXC, so an application which calls them will not compile until it is updated. PlaceOrder replaced its **positionSide** parameter with **openType**, and SetPositionLeverage replaced **marginMode**, which the endpoint does not accept, with **positionId**, **openType** and **positionType**. Several endpoint routes were corrected as well.

Method	Description
GetAccountAssets	Returns margin balances from **/api/v1/private/account/assets**.
GetPositionList	Lists current positions via **/api/v1/private/position/list/history_positions**.
SetPositionLeverage	Updates the leverage of a position through **/api/v1/private/position/change_leverage**. It takes the **symbol** and the **leverage**, plus an optional **positionId**, and **openType** (1 isolated, 2 cross) and **positionType** (1 long, 2 short) which are only required when the position is not open yet.
PlaceOrder	Places a futures order on **/api/v1/private/order/submit** providing **symbol**, **side**, **type**, **openType** (1 isolated, 2 cross), the quantity, which travels as **vol**, and the **price**, which the endpoint requires for market orders too, plus an optional **externalOid** and extra parameters.
CancelOrder	Cancels a specific order using **/api/v1/private/order/cancel** by **orderId** or **clientOrderId**.
CancelAllOrders	Cancels all open orders for a symbol via **/api/v1/private/order/cancel_all**.
GetOpenOrders	Lists current open orders through **/api/v1/private/order/list/open_orders** with an optional symbol filter.
GetOrderHistory	Retrieves order history (**/api/v1/private/order/list/history**) with optional symbol, time range and limit parameters.
GetFundingHistory	Returns past funding payments from **/api/v1/private/account/funding** with optional **startTime**, **endTime** and **limit** filters.

GetOpenPositions	Get all currently open positions.
ChangeMargin	Change the margin amount (ADD or SUB) of a position, pass the positionId of the position in aPositionId.
GetPositionMode	Get the current position mode (one-way or hedge mode).
ChangePositionMode	Change the position mode between one-way and hedge mode.
PlaceBatchOrder	Place multiple futures orders in a single batch request.
GetOrderDetail	Get detailed information for a specific order.
GetOrderDealDetails	Get fill/deal details for a specific order.
PlaceTriggerOrder	Place a trigger (plan) order that executes when conditions are met.
CancelTriggerOrder	Cancel a specific trigger order.
CancelAllTriggerOrders	Cancel all open trigger orders.
GetTriggerOrders	Get the list of trigger (plan) orders.
GetStopOrders	Get the list of stop orders.
CancelStopOrder	Cancel a specific stop order.
CancelAllStopOrders	Cancel all open stop orders.

API Bitget

Bitget

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Properties

Use the **BitgetChannel** property to select the channel type:
- **bgcSpot:** connect to the Spot WebSocket API.
- **bgcFutures:** connect to the Futures WebSocket API.

You can configure the following properties in the Bitget property.
- **ApiKey:** you can request a new api key in your Bitget account, just copy the value to this property. If the APIKey is set, the client will connect to the websocket private server. If it's empty, will connect to the Web-Socket public server.
- **ApiSecret:** it's the secret value of the api.
- **Passphrase:** it's the custom string defined when creating a new api key.
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Connection

When the client successfully connects to Bitget servers, the event **OnConnect** is fired. After the event **OnConnect** is fired, then you can start to **send** and **receive messages** to/from Bitget servers. If you are connecting to the private websocket channel, you must wait till **OnBitgetAuthentication** event is fired and check if the success parameter is true, before subscribing to any channel.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWS_API_Bitget *oBitget = new TsgcWS_API_Bitget();
oBitget->Client = oClient;
oBitget->Bitget->ApiKey = "your_api_key";
oBitget->Bitget->ApiSecret = "your_api_secret";
oBitget->Bitget->Passphrase = "your_passphrase";
oBitget->BitgetChannel = bgcSpot;
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
  DoLog("#Bitget Connected");
}
```

Events

The Bitget client implements the following events to control the connection flow and get data sent from the Web-Socket server:
- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnBitgetAuthentication:** fired when the client authenticates against the Private WebSocket Server.
- **OnBitgetSubscribe:** when the client subscribes to a websocket channel.
- **OnBitgetUnSubscribe:** when the client unsubscribes from a websocket channel.
- **OnBitgetData:** when the client receives data from the server.
- **OnBitgetError:** when there is any error during the Bitget websocket connection.
- **OnBitgetHTTPException:** when there is any error during the REST request.

WebSocket API

The websocket feed provides real-time market data updates for orders and trades.
You can subscribe to the following **channels**:

Method	Public or Private	Description
SubscribeTicker	Public	Subscribe to the ticker stream for an instrument.
SubscribeTrade	Public	Subscribe to the recent trades stream.
SubscribeCandle	Public	Subscribe to the candlestick stream. Supports multiple intervals (e.g. candle1m, candle5m, candle1H).
SubscribeOrderBook	Public	Subscribe to the order book stream. Supports different depth levels.
SubscribeOrders	Private	Subscribe to order updates. Requires authentication.
SubscribePositions	Private	Subscribe to position updates. Requires authentication.
SubscribeAccount	Private	Subscribe to account balance updates. Requires authentication.

Find below an example of subscribing to private websocket channels after a successful authentication.

```
void OnBitgetAuthentication(TObject *Sender, bool aSuccess, const string aError, const string aRawMessage)
{
  if (aSuccess == true)
  {
    oBitget->SubscribeOrders("BTCUSDT");
    oBitget->SubscribeAccount();
  }
}
```

REST API

The REST API provides Public and Private methods to request data from markets and private accounts. Access the REST API through the **REST_API** property of the component.

Method	Public / Private	Description
GetServerTime	Public	Get the server time.
GetTickers	Public	Get ticker information for one or all symbols.
GetOrderBook	Public	Get the order book for a symbol.
GetCandles	Public	Get candlestick/kline data for a symbol.
GetRecentTrades	Public	Get the most recent trades for a symbol.
PlaceOrder	Private	Place a new order.
CancelOrder	Private	Cancel an existing order.
GetOpenOrders	Private	Get the list of open orders.
GetOrderDetail	Private	Get details of a specific order.
GetOrderHistory	Private	Get the order history.
GetAccountAssets	Private	Get account asset information.
GetAccountInfo	Private	Get account information.

Find below an example of using the REST API.

```
TsgcWS_API_Bitget *oBitget = new TsgcWS_API_Bitget();
oBitget->Bitget->ApiKey = "your_api_key";
oBitget->Bitget->ApiSecret = "your_api_secret";
oBitget->Bitget->Passphrase = "your_passphrase";
// Get tickers
oBitget->REST_API->GetTickers("SPOT");
// Place an order
oBitget->REST_API->PlaceOrder("BTCUSDT", "buy", "market", "normal", "0.01");
```

API GateIO

Gate.io

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Properties

Use the **GateIOChannel** property to select the channel type:
- **giocSpot:** connect to the Spot WebSocket API.
- **giocFutures:** connect to the Futures WebSocket API.

You can configure the following properties in the GateIO property.
- **ApiKey:** you can request a new api key in your Gate.io account, just copy the value to this property. If the APIKey is set, the client will authenticate against the private server. If it's empty, only public channels will be available.
- **ApiSecret:** it's the secret value of the api.
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Connection

When the client successfully connects to Gate.io servers, the event **OnConnect** is fired. After the event **OnConnect** is fired, then you can start to **send** and **receive messages** to/from Gate.io servers. If you are connecting to the private websocket channel, you must wait till **OnGateIOAuthentication** event is fired and check if the success parameter is true, before subscribing to any private channel.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWS_API_GateIO *oGateIO = new TsgcWS_API_GateIO();
oGateIO->Client = oClient;
oGateIO->GateIO->ApiKey = "your_api_key";
oGateIO->GateIO->ApiSecret = "your_api_secret";
oGateIO->GateIOChannel = giocSpot;
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
  DoLog("#GateIO Connected");
}
```

Events

The GateIO client implements the following events to control the connection flow and get data sent from the WebSocket server:
- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnGateIOAuthentication:** fired when the client authenticates against the Private WebSocket Server.
- **OnGateIOSubscribe:** when the client subscribes to a websocket channel.
- **OnGateIOUnSubscribe:** when the client unsubscribes from a websocket channel.
- **OnGateIOData:** when the client receives data from the server.
- **OnGateIOError:** when there is any error during the GateIO websocket connection.
- **OnGateIOHTTPException:** when there is any error during the REST request.

WebSocket API

The websocket feed provides real-time market data updates for orders and trades.
You can subscribe to the following **channels**:

Method	Public or Private	Description
SubscribeTicker	Public	Subscribe to the ticker stream for a symbol.
SubscribeTrades	Public	Subscribe to the recent trades stream.
SubscribeCandlesticks	Public	Subscribe to the candlestick stream. Supports multiple intervals (e.g. 1m, 5m, 15m, 30m, 1h).
SubscribeOrderBook	Public	Subscribe to the order book stream. Supports configurable depth levels and update intervals.
SubscribeOrders	Private	Subscribe to order updates. Requires authentication.
SubscribeBalances	Private	Subscribe to balance updates. Requires authentication.

Find below an example of subscribing to public websocket channels.

```
void OnConnect(TsgcWSConnection *Connection)
{
  oGateIO->SubscribeTicker("BTC_USDT");
  oGateIO->SubscribeOrderBook("BTC_USDT", "20", "100ms");
}
```

REST API

The REST API provides Public and Private methods to request data from markets and private accounts. Access the REST API through the **REST_API** property of the component.

Method	Public / Private	Description
GetCurrencyPairs	Public	Get information about available currency pairs.
GetTickers	Public	Get ticker information for one or all currency pairs.
GetOrderBook	Public	Get the order book for a currency pair.
GetCandlesticks	Public	Get candlestick/kline data for a currency pair.
GetTrades	Public	Get the most recent trades for a currency pair.
PlaceOrder	Private	Place a new order. Supports limit and market orders.
CancelOrder	Private	Cancel an existing order.
GetOrder	Private	Get details of a specific order.
GetOpenOrders	Private	Get the list of open orders.
GetSpotAccounts	Private	Get spot account information.

Find below an example of using the REST API.

```
TsgcWS_API_GateIO *oGateIO = new TsgcWS_API_GateIO();
oGateIO->GateIO->ApiKey = "your_api_key";
oGateIO->GateIO->ApiSecret = "your_api_secret";
// Get tickers
oGateIO->REST_API->GetTickers("BTC_USDT");
// Place an order
oGateIO->REST_API->PlaceOrder("BTC_USDT", "buy", "0.001", "50000", "limit");
```

API Deribit

Deribit
Deribit is a cryptocurrency derivatives exchange offering futures and options trading on Bitcoin and Ethereum.

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Properties

You can configure the following properties in the Deribit property.

- **ApiKey:** you can request a new api key in your Deribit account, just copy the value to this property. If the APIKey is set, the client will authenticate against the private server. If it's empty, only public channels will be available.
- **ApiSecret:** it's the secret value of the api.
- **TestNet:** if enabled, will connect to the Deribit TestNet environment (disabled by default). Useful for testing without risking real funds.
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Connection

When the client successfully connects to Deribit servers, the event **OnConnect** is fired. After the event **OnConnect** is fired, then you can start to **send** and **receive messages** to/from Deribit servers. If you are connecting to the private websocket channel, you must wait till **OnDeribitAuthentication** event is fired and check if the success parameter is true, before subscribing to any private channel.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWS_API_Deribit *oDeribit = new TsgcWS_API_Deribit();
oDeribit->Client = oClient;
oDeribit->Deribit->ApiKey = "your_api_key";
oDeribit->Deribit->ApiSecret = "your_api_secret";
oDeribit->Deribit->TestNet = false;
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
  DoLog("#Deribit Connected");
}
```

Events

The Deribit client implements the following events to control the connection flow and get data sent from the WebSocket server:

- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnDeribitAuthentication:** fired when the client authenticates against the Private WebSocket Server.
- **OnDeribitSubscribe:** when the client subscribes to a websocket channel.
- **OnDeribitUnSubscribe:** when the client unsubscribes from a websocket channel.
- **OnDeribitData:** when the client receives data from the server.
- **OnDeribitError:** when there is any error during the Deribit websocket connection.
- **OnDeribitHTTPException:** when there is any error during the REST request.

WebSocket API

The websocket feed provides real-time market data updates for orders and trades.
You can subscribe to the following **Public channels**:

Method	Description
SubscribeTicker	Subscribe to the ticker stream for an instrument. Provides real-time price and volume data.
SubscribeTrades	Subscribe to the recent trades stream for an instrument.
SubscribeOrderBook	Subscribe to the order book stream. Supports configurable grouping, depth levels and update intervals.
SubscribeCandle	Subscribe to the candlestick stream. Supports multiple resolutions (e.g. 1, 3, 5, 10, 15, 30, 60, 120, 180, 360, 720, 1D).
SubscribePerpetual	Subscribe to perpetual contract data for an instrument. Provides funding rate and other perpetual-specific information.
SubscribeBookChange	Subscribe to order book change notifications for an instrument.

You can subscribe to the following **Private channels** (require authentication):

Method	Description
SubscribeOrders	Subscribe to order updates for an instrument.
SubscribePositions	Subscribe to position updates. Supports filtering by currency and instrument kind.
SubscribeAccountChanges	Subscribe to account balance changes for a currency.
SubscribeUserTrades	Subscribe to user trade updates for an instrument.

Find below an example of subscribing to websocket channels.

```
void OnDeribitAuthentication(TObject *Sender, bool aSuccess, const string aError, const string aRawMessage)
{
  if (aSuccess == true)
  {
    oDeribit->SubscribeOrders("BTC-PERPETUAL");
    oDeribit->SubscribePositions("BTC");
  }
}
```

REST API

The REST API provides Public and Private methods to request data from markets, trading, account, and wallet. Access the REST API through the **REST_API** property of the component.

Market Data (Public)

Method	Description
GetInstruments	Get available instruments for a currency.
GetTicker	Get ticker information for an instrument.
GetOrderBook	Get the order book for an instrument.
GetTrades	Get the most recent trades for an instrument.
GetCurrencies	Get the list of supported currencies.
GetIndexPrice	Get the current index price for an index name.
GetFundingRateHistory	Get the funding rate history for an instrument.
GetFundingRateValue	Get the current funding rate value for an instrument.
GetBookSummaryByCurrency	Get book summary for all instruments of a currency.
GetBookSummaryByInstrument	Get book summary for a specific instrument.

Trading (Private)

Method	Description
Buy	Place a buy order. Supports market and limit order types.

Sell	Place a sell order. Supports market and limit order types.
CancelOrder	Cancel an existing order by order ID.
CancelAllOrders	Cancel all open orders.
CancelAllByInstrument	Cancel all open orders for a specific instrument.
EditOrder	Edit an existing order (change amount or price).
GetOpenOrders	Get the list of open orders for a currency.
GetOrderState	Get the state of a specific order.
GetOrderHistory	Get the order history for a currency.

Account (Private)

Method	Description
GetPositions	Get positions for a currency.
GetAccountSummary	Get the account summary for a currency.
GetSubAccounts	Get the list of sub-accounts.
GetTransactionLog	Get the transaction log for a currency.

Wallet (Private)

Method	Description
GetDeposits	Get the deposit history for a currency.
GetWithdrawals	Get the withdrawal history for a currency.
GetTransfers	Get the transfer history for a currency.

Find below an example of using the REST API.

```
TsgcWS_API_Deribit *oDeribit = new TsgcWS_API_Deribit();
oDeribit->Deribit->ApiKey = "your_api_key";
oDeribit->Deribit->ApiSecret = "your_api_secret";
// Get ticker
oDeribit->REST_API->GetTicker("BTC-PERPETUAL");
// Place a buy order
oDeribit->REST_API->Buy("BTC-PERPETUAL", 10, "market");
// Get account summary
oDeribit->REST_API->GetAccountSummary("BTC");
```

API Crypto.com

Crypto.com

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Properties

Use the **Channel** property to select the channel type:
- **cccMarket:** connect to the Market WebSocket API for public market data.
- **cccUser:** connect to the User WebSocket API for private account data (requires authentication).

You can configure the following properties in the CryptoCom property.
- **ApiKey:** you can request a new api key in your Crypto.com account, just copy the value to this property. Required for private channels and authenticated REST API calls.
- **ApiSecret:** it's the secret value of the api.
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Connection

When the client successfully connects to Crypto.com servers, the event **OnConnect** is fired. After the event **OnConnect** is fired, then you can start to **send** and **receive messages** to/from Crypto.com servers. If you are connecting to the User channel, you must wait till **OnCryptoComAuthentication** event is fired and check if the success parameter is true, before subscribing to any private channel.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWS_API_CryptoCom *oCryptoCom = new TsgcWS_API_CryptoCom();
oCryptoCom->Client = oClient;
oCryptoCom->CryptoCom->ApiKey = "your_api_key";
oCryptoCom->CryptoCom->ApiSecret = "your_api_secret";
oCryptoCom->Channel = cccMarket;
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
  DoLog("#CryptoCom Connected");
}
```

Events

The Crypto.com client implements the following events to control the connection flow and get data sent from the WebSocket server:
- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnCryptoComAuthentication:** fired when the client authenticates against the Private WebSocket Server.
- **OnCryptoComSubscribe:** when the client subscribes to a websocket channel.
- **OnCryptoComUnSubscribe:** when the client unsubscribes from a websocket channel.
- **OnCryptoComData:** when the client receives data from the server.
- **OnCryptoComError:** when there is any error during the Crypto.com websocket connection.
- **OnCryptoComHTTPException:** when there is any error during the REST request.

WebSocket API

The websocket feed provides real-time market data updates for orders and trades.
You can subscribe to the following **Market channels** (public):

Method	Description
SubscribeTicker	Subscribe to the ticker stream for an instrument. Provides real-time price and volume data.
SubscribeTrade	Subscribe to the recent trades stream for an instrument.
SubscribeCandlestick	Subscribe to the candlestick stream. Supports multiple intervals (e.g. 5m, 15m, 30m, 1h, 4h, 1D).
SubscribeBook	Subscribe to the order book stream. Supports configurable depth levels (e.g. 10, 50).

You can subscribe to the following **User channels** (private, require authentication):

Method	Description
SubscribeOrders	Subscribe to order updates. Optionally filter by instrument name.
SubscribeUserTrades	Subscribe to user trade updates. Optionally filter by instrument name.
SubscribeBalance	Subscribe to balance updates for your account.

Find below an example of subscribing to market channels.

```
void OnConnect(TsgcWSConnection *Connection)
{
  oCryptoCom->SubscribeTicker("BTC_USDT");
  oCryptoCom->SubscribeBook("BTC_USDT", "10");
}
```

REST API

The REST API provides Public and Private methods to request data from markets and private accounts. Access the REST API through the **REST_API** property of the component.

Method	Public / Private	Description
GetInstruments	Public	Get the list of available instruments.
GetTicker	Public	Get ticker information for one or all instruments.
GetOrderBook	Public	Get the order book for an instrument.
GetTrades	Public	Get the most recent trades for an instrument.
GetCandlestick	Public	Get candlestick/kline data for an instrument.
CreateOrder	Private	Create a new order. Supports limit and market orders.
CancelOrder	Private	Cancel an existing order.
CancelAllOrders	Private	Cancel all open orders for an instrument.
GetOpenOrders	Private	Get the list of open orders.
GetOrderDetail	Private	Get details of a specific order.
GetOrderHistory	Private	Get the order history.
GetAccountSummary	Private	Get account summary information.

Find below an example of using the REST API.

```
TsgcWS_API_CryptoCom *oCryptoCom = new TsgcWS_API_CryptoCom();
oCryptoCom->CryptoCom->ApiKey = "your_api_key";
oCryptoCom->CryptoCom->ApiSecret = "your_api_secret";
// Get instruments
oCryptoCom->REST_API->GetInstruments();
// Create an order
oCryptoCom->REST_API->CreateOrder("BTC_USDT", "BUY", "MARKET", "", "0.001");
// Get account summary
oCryptoCom->REST_API->GetAccountSummary();
```

API HTX

HTX
HTX (formerly Huobi) is an international multi-language cryptocurrency exchange. The HTX API component provides an HTTP REST API to complement the existing Huobi WebSocket API.

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more. See the Huobi API documentation for WebSocket subscription methods.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Properties

You can configure the following properties in the Huobi property (for WebSocket) or HTXOptions property (for REST API).

- **ApiKey:** you can request a new api key in your HTX account, just copy the value to this property. If the APIKey is set, the client will connect to the websocket private server. If it's empty, will connect to the WebSocket public server.
- **ApiSecret:** it's the secret value of the api.
- **Resubscribe:** if enabled, after a reconnection the client automatically subscribes again to the channels that were active before the disconnection (by default false).

Connection

The HTX component extends the Huobi WebSocket API client. When the client successfully connects to HTX servers, the event **OnConnect** is fired. If the ApiKey is not empty, the client will attempt to connect to the private websocket server, so only the private methods will be available. If the ApiKey is empty, the client will connect to the public websocket server and only the public methods will be available.

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWS_API_Huobi *oHTX = new TsgcWS_API_Huobi();
oHTX->Client = oClient;
oHTX->Huobi->ApiKey = "your_api_key";
oHTX->Huobi->ApiSecret = "your_api_secret";
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
  DoLog("#HTX Connected");
}
```

Events

The HTX client implements the following events to control the connection flow and get data sent from the WebSocket server:

- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnHuobiSubscribed:** event called after a successful subscription.
- **OnHuobiUnSubscribed:** event called after a successful unsubscription.
- **OnHuobiUpdate:** every time there is an update in data (kline, market depth...) this event is called.
- **OnHuobiError:** if there is an error in the HTX API, this event will provide information about the error.

WebSocket API

The WebSocket API provides real-time market data updates. See the Huobi API documentation for the full list of WebSocket subscription methods including public channels (SubscribeKLine, SubscribeMarketDepth, SubscribeTradeDetail, etc.) and private channels (SubscribeOrderUpdates, SubscribeTradeClearing, SubscribeAccountChange).

REST API

The REST API provides Public and Private methods to request data from markets and private accounts. The REST API is accessed through the **TsgcHTTP_API_HTX** component.

Market Data (Public)

Method	Description
GetServerTime	Get the server time.
GetSymbols	Get the list of available trading symbols.
GetCurrencys	Get the list of supported currencies.
GetMarketTickers	Get tickers for all trading symbols.
GetMarketDetail	Get the 24-hour market detail for a symbol.
GetMarketDepth	Get the order book depth for a symbol. Supports configurable depth type and depth levels.
GetMarketTrade	Get the most recent trade for a symbol.
GetMarketHistory-Trade	Get the most recent trades history for a symbol.
GetMarketHistoryKline	Get candlestick/kline data. Supports multiple periods (1min, 5min, 15min, 30min, 60min, 4hour, 1day, 1mon, 1week, 1year).

Trading (Private)

Method	Description
PlaceOrder	Place a new order. Supports buy-market, sell-market, buy-limit, sell-limit and other order types.
CancelOrder	Cancel an existing order by order ID.
GetOrder	Get details of a specific order.
GetOpenOrders	Get the list of open orders for an account.
GetOrderHistory	Get the order history for a symbol.

Account (Private)

Method	Description
GetAccounts	Get the list of accounts.
GetAccountBalance	Get the balance of a specific account.
GetAccountAssetValuation	Get the total asset valuation for an account type (e.g. spot).

Find below an example of using the REST API.

```
TsgcHTTP_API_HTX *oHTX = new TsgcHTTP_API_HTX();
oHTX->HTXOptions->ApiKey = "your_api_key";
oHTX->HTXOptions->ApiSecret = "your_api_secret";
// Get market tickers
oHTX->GetMarketTickers();
// Get kline data
oHTX->GetMarketHistoryKline("btcusdt", "1min", 150);
// Place an order
oHTX->PlaceOrder("12345678", "btcusdt", "buy-limit", "0.001", "50000");
// Get account balance
oHTX->GetAccountBalance("12345678");
```

WhatsApp Cloud API

Whatsapp

Send and **receive messages** using a cloud-hosted version of the **WhatsApp Business Platform**. The **Cloud API** allows you to implement WhatsApp Business APIs without the cost of hosting of your own servers and also allows you to more easily scale your business messaging. The Cloud API supports up to 80 messages per second of combined sending and receiving (inclusive of text and media messages).

The WhatsApp Business API allows medium and large businesses to communicate with their customers at scale. Using the API, businesses can build systems that connect thousands of customers with agents or bots, enabling both programmatic and manual communication. Additionally, you can integrate the API with numerous backend systems, such as CRM and marketing platforms.

Features

Businesses will get all the new features faster on Cloud API. Right now, WhatsApp Business Cloud API comes with all the features that are available with WhatsApp Business API.

Useful features of WhatsApp Cloud API:

- **Integrate** WhatsApp messaging with tools like **CRM**, **analytics**, and **third-party** apps
- **Green Tick**, verified WhatsApp Business profile
- WhatsApp **Broadcast** & **Bulk Messaging**
- No app or interface, use via BSPs or CRM
- **WhatsApp Chatbot** & chat **automation** using third-party apps
- **Schedule** WhatsApp messages at a large scale
- **Interactive messaging** features include List messages, reply buttons, CTA messages

Most common uses

- **Configuration**
  - **WhatsApp Create App**
  - **WhatsApp Phone Number Id**
  - **WhatsApp Token**
  - **WhatsApp Webhook**
  - **WhatsApp Security**

- **Messages**
  - **WhatsApp Send Messages**
  - **WhatsApp Send Interactive Messages**
  - **WhatsApp Send Template Messages**
  - **WhatsApp Receive Messages and Status Notifications**
  - **WhatsApp Send Files**
  - **WhatsApp Download Media**

Get Started

To send and receive a first message using a test number, complete the following steps:

1. Set up Developer Assets and Platform Access

- Register as a Meta Developer
- Enable two-factor authentication for your account

- Create a Meta App: Go to **developers.facebook.com** > **My Apps** > **Create App**. Select the "Business" type and follow the prompts on your screen.

From the App Dashboard, click on the app you would like to connect to WhatsApp. Scroll down to find the "WhatsApp" product and click **Set up**.

Next, you will see the option to select an existing Business Manager (if you have one) or, if you would like, the onboarding process can create one automatically for you (you can customize your business later, if needed). Make a selection and click **Continue**.

When you click **Continue**, the onboarding process performs the following actions:

- Your App is associated with the Business Manager that you chose, or that was created automatically.
- A WhatsApp test phone number is added to your business. You can use this test phone number to explore the WhatsApp Business Platform without registering or migrating a real phone number. Test phone numbers can send unlimited messages to up to 5 recipients (which can be anywhere in the world).

2. Send a Test Message

Now, you can open your IDE and create a new project. Drop a TsgcWhatsapp_Client component and fill the following properties:

- **WhatsappOptions.PhoneNumberId:** is the ID of the Phone Number used to send messages.
- **WhatsappOptions.Token:** is the Temporary Access Token valid for 24 hours.

Once those 2 properties have been properly configured, call the method **SendTest** to send your **First message** to a phone number using the **Whatsapp Business Platform**.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendTest("34605889421");
```

3. Configure a Webhook

To get alerted when you receive a message or when a message's status has changed, you need to set up a Webhooks endpoint for your app. Setting up Webhooks doesn't affect the status of your phone number and does not interfere with you sending or receiving messages.

To get started, first you need to create the endpoint, so first configure the **ServerOptions** property of WhatsApp Client component and configure the following properties:

- **ServerOptions:** here you can configure the IP Address to bind, the Listening Port, if it's using SSL (the WebHook must run in a secure server, you can configure your server to use SSL or Proxy the WebHook requests to a none HTTPs server). The server is based on TsgcWebSocketHTTPServer.
  - **WebhookOptions:** this property allows you to set the Webhook properties that later will be configured in your developer facebook account.
    - **Endpoint:** it's the name of the endpoint, by default is /webhook. Example: if your server is listening on https://www.esegece.com, the endpoint will be "https://www.esegece.com/webhook"
    - **Token:** it's a security string that can contain any value defined by you. It's used to verify the Webhook registration is correct.

After configuring the server, you can use the method **StartServer** to start the server and accept the incoming requests.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->ServerOpptions->WebhookOptions->PhoneNumberId = "/webhook";
oClient->ServerOptions->WebhookOptions->Token = "MySecretToken";
oClient->StartServer();
```

Once your endpoint is ready, go to your App Dashboard.

In your App Dashboard, find the WhatsApp product and click **Configuration**. Then, find the webhooks section and click **Configure a webhook**. After the click, a dialog appears on your screen and asks you for two items:

- Callback URL: This is the URL Meta will be sending the events to.
- Verify Token: This string is set up by you, when you create your webhook endpoint.

After adding the information, click **Verify and Save**.

Back in the App Dashboard, click **WhatsApp** > **Configuration** in the left-side panel. Under Webhooks, click **Manage**. A dialog box will open with all the objects you can get notified about. To receive messages from your users, click **Subscribe** for **messages**.

4. Receive a test message

Every time a new message is received, the client event **OnMessageReceived** will be called.

```
void OnMessageReceived(TObject *Sender, TsgcWhatsapp_Receive_Message *aMessage, ref bool aMarkAsRead)
{
  DoLog("Received: " + aMessage->Messages->_Message[0]->Id);
}
```

Now that your Webhook is set up, send a message to the test number you have used. You should immediately get a Webhooks notification with the content of your message!

The WhatsApp API does not allow sending free text messages to phones that have not contacted you before (within the latest 24 hours). The only way to send a text message to a phone that has never texted your developer account number is by sending a Template (previously approved by Meta). To override this limitation for testing free text messages, first send a WhatsApp message from the destination number to your developer account number, and then you will be able to send free text messages for 24 hours.

Events

OnBeforeSendMessage

This event is called before the message is sent to the WhatsApp servers. You can access the internal message through the RawMessage parameter.

OnBeforeSubscribe

This event is called before the server subscribes to a topic. Use the Accept parameter to allow or deny the subscription. By default, the server will subscribe to all events requested.

OnRawMessage

This event is called when the server receives a new message that has not yet been parsed, so you get access to the raw message.

OnMessageReceived

This event is called after the server receives and parses a new message. If you set the MarkAsRead parameter to True, the sender will receive a double check.

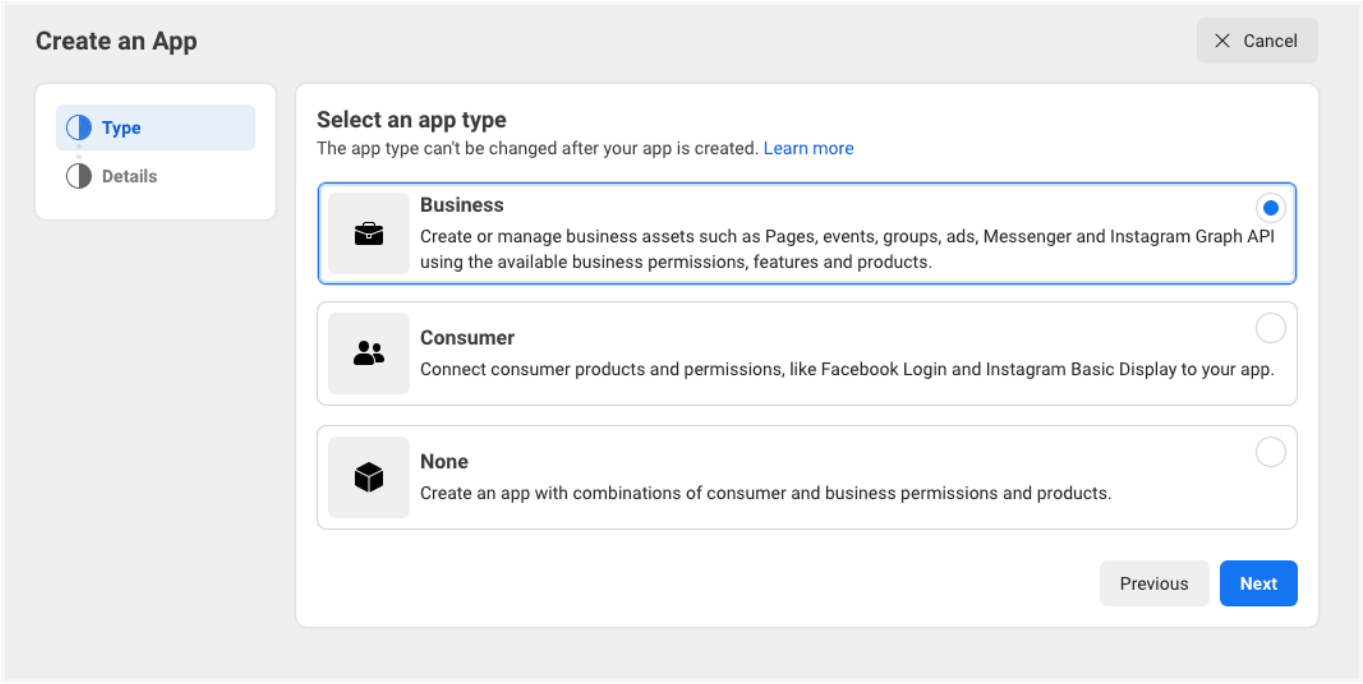
OnMessageSent

This event is called every time the server receives a new status message about the message previously sent. Read the Status property to know if the message has been sent, delivered or read.

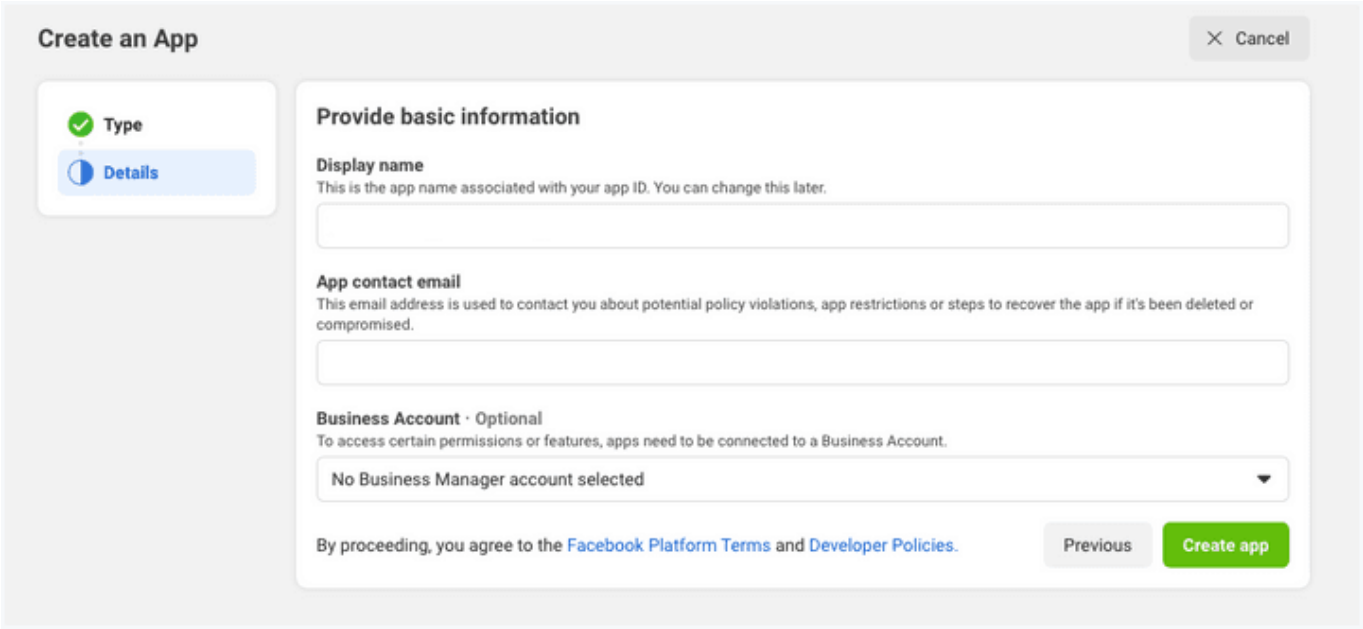
WhatsApp Create App

Go to developers.facebook.com and **Create App**.

Select **Business Type** as the app type and proceed.



Provide a name for your app (avoid using trademarked names such as “WhatsApp” or “Facebook”).



Once the app has been created, click the **WhatsApp button** on the next screen to add WhatsApp sending capabilities to your app.

On the next screen, you will be required to link your WhatsApp app to your Facebook business account. You will also have the option to create a new business account if you don’t have one yet.

WhatsApp Phone Number Id

When you register with WhatsApp Cloud API, Facebook provides a Test WhatsApp phone number that will be the default sending address of your Application. For recipients, you will have the option to add a maximum of 5 phone numbers during the development phase without having to make any payment.

Later you can register your own Phone Number to avoid the limitation of 5 phone numbers.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
```

WhatsApp Token

WhatsApp Cloud API requires a valid token to send any message using the Cloud API.

Facebook provides a Test WhatsApp phone number that allows you to send messages up to 5 phone numbers. You can override later this limitation registering your own phone number.

The WhatsApp API provides a **Temporary Access Token** that will be valid for 23 hours. This token must be configured in the TsgcWhatsApp_Client component to allow sending messages.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
```

If you need a long-valid token, you can create (or update) a system user and generate a new token with the **whatsapp_business_messaging** permission. This will allow you to send and receive WhatsApp messages without updating the token every 23 hours.

WhatsApp Webhook

Subscribe to Webhooks to get notifications about messages your business receives and customer profile updates.

Create Endpoint

Before you can start receiving notifications you will need to create an endpoint on your server to receive notifications.

Your endpoint must be able to process two types of HTTPS requests: Verification Requests and Event Notifications. Since both requests use HTTPs, your server must have a valid TLS or SSL certificate correctly configured and installed. Self-signed certificates are not supported.

When you configure the Webhook in the WhatsApp Settings, you must define the endpoint where is listening your server and a Token that can be any value, this token is used when registering the webhook endpoint and verify the subscriber is valid.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->ServerOpptions->WebhookOptions->PhoneNumberId = "/webhook";
oClient->ServerOptions->WebhookOptions->Token = "MySecretToken";
oClient->StartServer();
```

Once the Webhook is configured, subscribe to **Messages** Webhook Fields to be notified every time a new message is received.

You can read more about configuring SSL Server.

WhatsApp Security

Every time a new message is received or there is a new status of a message, the server receives a notification in the endpoint configured in the Webhook. To be sure the request comes from WhatsApp Cloud API Servers, the request contains a header with a signature, you can configure the WhatsApp client to verify the signatures before process the message.

To do this, first you need to set the Application Secret in the property **ServerOptions.Application.Secret** and enable **VerifySignature** property.

Once configured, every time a new message is received, first the signature is verified, and if it's wrong, returns an error 500 and the message is not processed.

WhatsApp Send Messages

All API calls must be authenticated with an Access Token. Developers can authenticate their API calls with the access token generated in **App Dashboard** > **WhatsApp** > **Getting Started**

The API calls return the Message Id as a string.

Text Messages

Call the method **SendMessageText** and pass the following parameters:

- **aTo:** phone number
- **aText:** text of the message.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageText("34605889421", "Hello from sgcWebSockets!!!");
```

Image Messages

Call the method **SendMessageImage** and pass the following parameters:

- **aTo:** phone number
- **aLink:** url where is the image to send
- **aCaption:** title of the image (optional).

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageImage("34605889421", "https://www.media.com/image.png", "logo");
```

Document Messages

Call the method **SendMessageDocument** and pass the following parameters:

- **aTo:** phone number
- **aLink:** url where is the document to send
- **aCaption:** title of the document (optional).
- **aFileName:** name of the file (optional).

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageDocument("34605889421", "https://www.documents.com/file.txt", "Document", "file.txt");
```

Audio Messages

Call the method **SendMessageAudio** and pass the following parameters:

- **aTo:** phone number
- **aLink:** url where is the audio to send

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageAudio("34605889421", "https://www.audio.com/audio.mp3");
```

Video Messages

Call the method **SendMessageVideo** and pass the following parameters:

- **aTo:** phone number
- **aLink:** url where is the video to send

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageVideo("34605889421", "https://www.video.com/audio.mp4");
```

Sticker Messages

Call the method **SendMessageSticker** and pass the following parameters:

- **aTo:** phone number
- **aLink:** url where is the sticker to send

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageSticker("34605889421", "https://www.stickers.com/sticker");
```

Location Messages

Call the method **SendMessageLocation** and pass the following parameters:

- **aTo:** phone number
- **aLongitude:** Longitude of the location.
- **aLatitude:** Latitude of the location.
- **aName:** Name of the location.
- **aAddress:** Address of the location. Only displayed if aName is set.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageLocation("34605889421", "50.159305", "9.762686", "My Location", "My Address");
```

Contact Messages

Call the method **SendMessageContact** and pass the following parameters:

- **aTo:** phone number
- **aName:** Full name, as it normally appears (required).
- **aPhone:** the phone number (optional).
- **aEmail:** the email (optional).

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageLocation("34605889421", "John Smith", "15550386570", "john@mail.com");
```

WhatsApp Send Interactive Messages

Interactive messages give your users a simpler way to find and select what they want from your business on WhatsApp. During testing, chatbots using interactive messaging features achieved significantly higher response rates and conversions compared to those that are text-based.

The following messages are considered interactive:

- **List Messages**: Messages including a menu of up to 10 options. This type of message offers a simpler and more consistent way for users to make a selection when interacting with a business.
- **Reply Buttons**: Messages including up to 3 options —each option is a button. This type of message offers a quicker way for users to make a selection from a menu when interacting with a business. Reply buttons have the same user experience as interactive templates with buttons.

Interactive Message Specifications

- Interactive messages can be combined together in the same flow.
- Users cannot select more than one option at the same time from a list or button message, but they can go back and re-open a previous message.
- List or reply button messages cannot be used as notifications. Currently, they can only be sent within 24 hours of the last message sent by the user. If you try to send a message outside the 24-hour window, you get an error message.

When You Should Use It

List Messages are best for presenting several options, such as:

- A customer care or FAQ menu
- A take-out menu
- Selection of nearby stores or locations
- Available reservation times
- Choosing a recent order to repeat

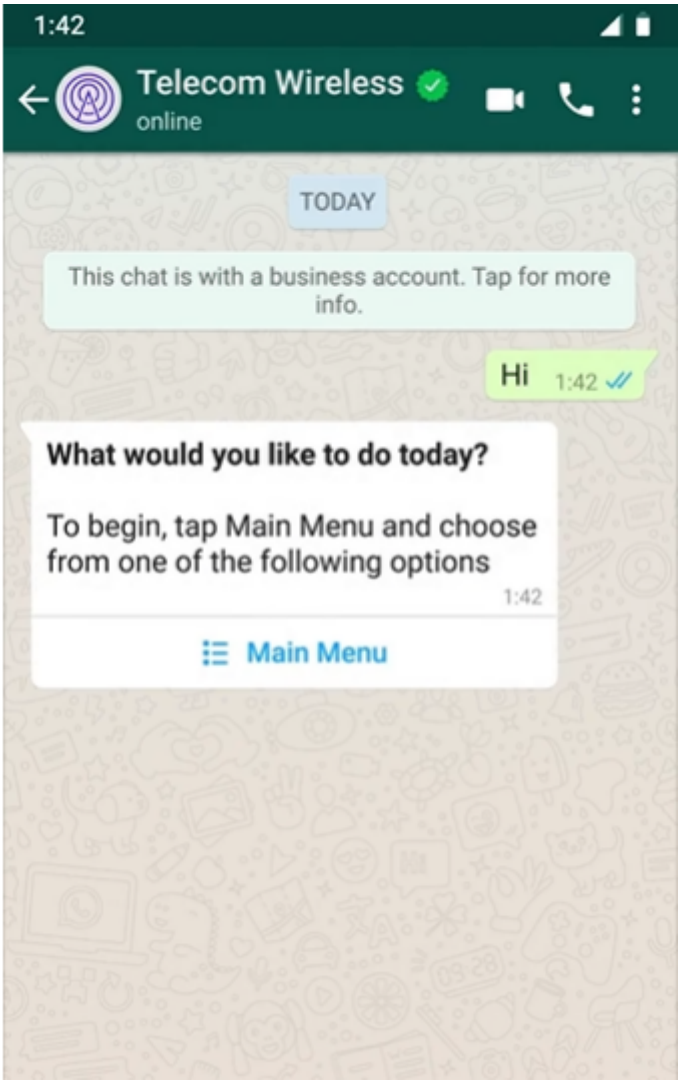
Reply Buttons are best for offering quick responses from a limited set of options, such as:

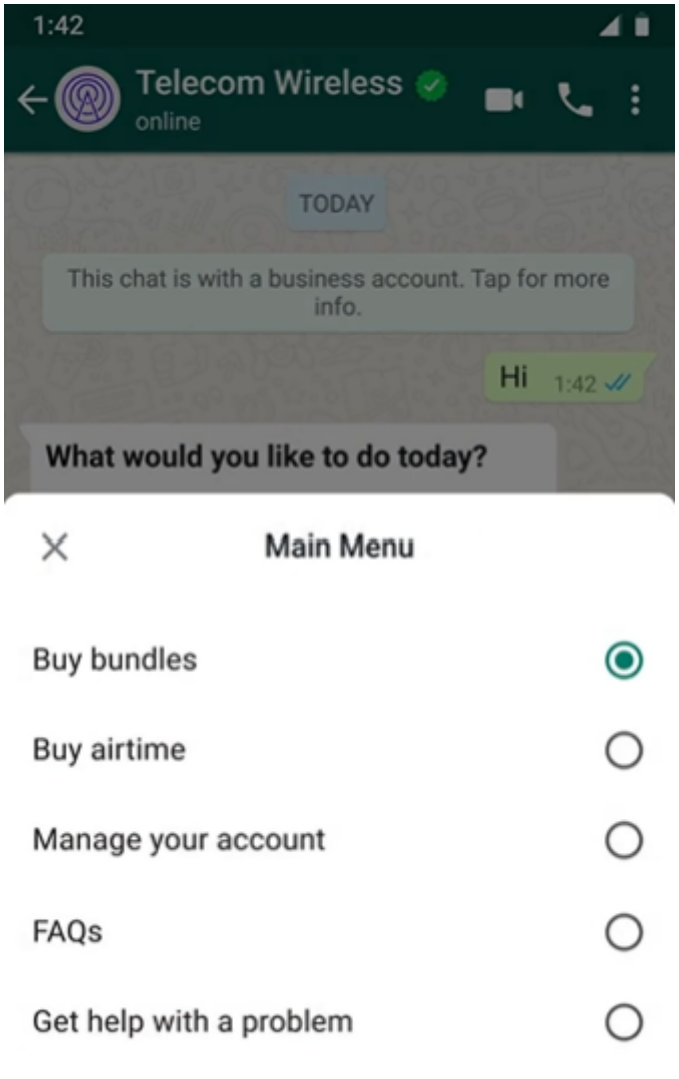
- Airtime recharge
- Changing personal details
- Reordering a previous order
- Requesting a return
- Adding optional extras to a food order
- Choosing a payment method

Reply buttons are particularly valuable for ‘personalized’ use cases where a generic response is not adequate.

Interactive List

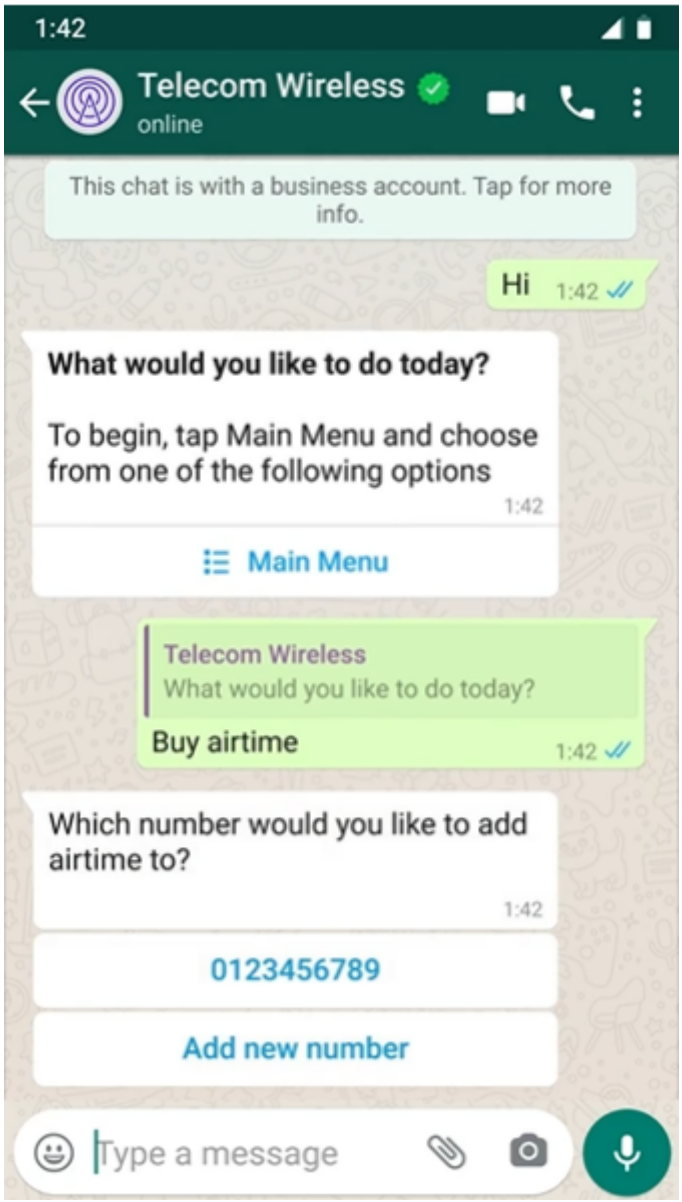
```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageInteractiveList("

34605889421

", "What Would you like to do today?", "To begin, Tap Main Menu and choose from of the following options", "", "M
```

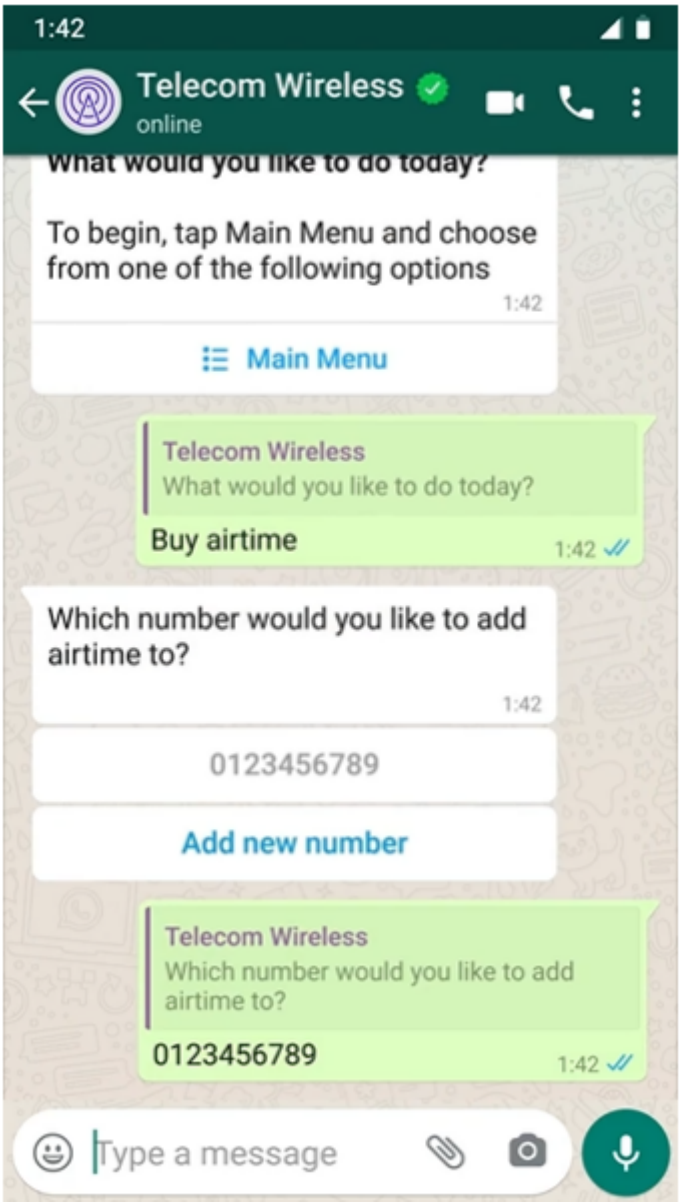




Reply Buttons

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageInteractiveButtons("34605889421", "Select an option", "Which number would you like to add air
```


1:42
Telecom Wireless
online
This chat is with a business account. Tap for more info.
Hi 1:42
What would you like to do today?
To begin, tap Main Menu and choose from one of the following options
1:42
Main Menu
Telecom Wireless
What would you like to do today?
Buy airtime 1:42
Which number would you like to add airtime to?
1:42
0123456789
Add new number
Type a message


1:42
Telecom Wireless
online
What would you like to do today?
To begin, tap Main Menu and choose from one of the following options
1:42
Main Menu
Telecom Wireless
What would you like to do today?
Buy airtime
1:42
Which number would you like to add airtime to?
1:42
0123456789
Add new number
Telecom Wireless
Which number would you like to add airtime to?
0123456789
1:42
Type a message

WhatsApp Send Template Messages

Call the method **SendMessageTemplate** and pass the following parameters:

- **aTo:** phone number
- **aTemplate:** template identifier.
- **aLanguageCode:** template language.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageTemplate("34605889421", "hello_world", "en_US");
```

Template Message Parameters

Templates can include parameters, see below an example of default template with parameters

```
void SendSamplePurchaseFeedbackTemplate(string aName)
{
  TsgcWhatsApp_Send_Message_Template *oTemplate = new TsgcWhatsApp_Send_Message_Template();
  Try
    oTemplate->Language->Code = "en_US";
    oTemplate->TemplateName = "sample_purchase_feedback";
    // ... header
    TsgcWhatsApp_Send_Message_Template_Component *oComponent = new TsgcWhatsApp_Send_Message_Template_Component()
    oComponent->_Type = wapctHeader;
    oTemplate->Components->Add(oComponent);
    TsgcWhatsApp_Send_Message_Template_Parameter *oParameter = new TsgcWhatsApp_Send_Message_Template_Parameter()
    oParameter->Image->Link = "https://www.esegece.com/images/esegece.png";
    oParameter->_Type = wapptImage;
    oComponent->Parameters->Add(oParameter);
    // ... body
    TsgcWhatsApp_Send_Message_Template_Component *oComponent = new TsgcWhatsApp_Send_Message_Template_Component()
    oComponent->_Type = wapctBody;
    oTemplate->Components->Add(oComponent);
    TsgcWhatsApp_Send_Message_Template_Parameter *oParameter = new TsgcWhatsApp_Send_Message_Template_Parameter()
    oParameter->Text = aName;
    oParameter->_Type = wapptText;
    oComponent->Parameters->Add(oParameter);
    whatsapp->SendMessageTemplate("107809351952205", oTemplate);
  __Finally
    oTemplate->Free();
  End;
}
```

Template Message Uploaded Image

Find below an example of a template where instead of using a link to an image, first uploads the image to the server and then sets the Id of the document.

```
oClient->OnMessage(TsgcWSConnection *Connection, const string Text)
void SendSamplePurchaseFeedbackTemplate(string aName)
{
  TsgcWhatsApp_Send_Message_Template *oTemplate = new TsgcWhatsApp_Send_Message_Template();
  Try
    // ... first upload the file
    string vId = whatsapp->UploadMedia("c:\images\file.png", "image/png");

    // ... send message
    oTemplate->Language->Code = "en_US";
    oTemplate->TemplateName = "sample_purchase_feedback";
    // ... header
    TsgcWhatsApp_Send_Message_Template_Component *oComponent = new TsgcWhatsApp_Send_Message_Template_Component()
```

```
    oComponent->_Type = wapctHeader;
    oTemplate->Components->Add(oComponent);
    TsgcWhatsApp_Send_Message_Template_Parameter *oParameter = new TsgcWhatsApp_Send_Message_Template_Parameter()
    oParameter->Image->id = vId;
    oParameter->_Type = wapptImage;
    oComponent->Parameters->Add(oParameter);
    // ... body
    TsgcWhatsApp_Send_Message_Template_Component *oComponent = new TsgcWhatsApp_Send_Message_Template_Component()
    oComponent->_Type = wapctBody;
    oTemplate->Components->Add(oComponent);
    TsgcWhatsApp_Send_Message_Template_Parameter *oParameter = new TsgcWhatsApp_Send_Message_Template_Parameter()
    oParameter->Text = aName;
    oParameter->_Type = wapptText;
    oComponent->Parameters->Add(oParameter);
    whatsapp->SendMessageTemplate("107809351952205", oTemplate);
  __Finally
    oTemplate->Free();
  End;
}
```

WhatsApp Receive Messages and Status Notifications

Subscribe to Webhooks to get notifications about messages your business receives and customer profile updates.

Whenever a trigger event occurs, the WhatsApp Business Platform sees the event and sends a notification to a Webhook URL you have previously specified. You can get two types of notifications:

- **Received messages**: This alert lets you know when you have received a message.
- **Message status and pricing notifications**: This alert lets you know when the status of a message has changed —for example, the message has been read or delivered.

Received Messages

Every time a new message is received the event **OnMessageReceived** is called, where you can access to the content of the Message and mark the message as read.

Find below an example, when a new text message is received, it's echoed to user who sent it.

```
void OnWhatsAppMessageReceived(TObject *Sender, const TsgcWhatsApp_Receive_Message *aMessage, ref bool aMarkAsRea
{
  if (aMessage->Contacts->Count > 0)
  {
    string vTo = aMessage->Contacts->Contact[0]->WaID;
    if (aMessage->Messages->Count > 0)
    {
      if (aMessage->Messages->_Message[0]->_Type = wapmrtText)
      {
        vText = "ECHO ==> " + aMessage->Messages->_Message[0]->Text->Body;
        WhatsApp->SendMessageText(vTo, vText);
        aMarkAsRead = true;
      }
    }
  }
}
```

Sent Messages

The WhatsApp Business Platform sends notifications to inform you of the status of the messages between you and users. When a message is sent successfully, you receive a notification when the message is sent, delivered, and read. The order of these notifications in your app may not reflect the actual timing of the message status. View the timestamp to determine the timing, if necessary.

- **sent:** The following notification is received when a business sends a message as part of a user-initiated conversation (if that conversation did not originate in a free entry point):
- **delivered:** The following notification is received when a business' message is delivered and that message is part of a user-initiated conversation (if that conversation did not originate in a free entry point):
- **read:** The following notification is received when the user reads the message.

Every time a new status is received, the event **OnMessageSent** is called.

```
void OnWhatsAppMessageSent(TObject *Sender, const TsgcWhatsApp_Receive_Message *aMessage, TsgcWhatsAppSendMessage
{
```

```
  string vPhone := aMessage.MetaData.DisplayPhoneNumber
  if (aStatus = wapsmstSent) {
    DoLog("Message to " + vPhone + " sent.");
  } else if (aStatus = wapsmstDelivered) {
    DoLog("Message to " + vPhone + " delivered.");
  } else if (aStatus = wapsmstRead) {
    DoLog("Message to " + vPhone + " read.");
  } else {
    DoLog("Message to " + vPhone + " unknown status.");
  }
}
```

WhatsApp Send Files

All API calls must be authenticated with an Access Token. Developers can authenticate their API calls with the access token generated in **App Dashboard** > **WhatsApp** > **Getting Started**

The API calls return the Message Id as a string.

When you send a File using the WhatsApp API, first the message is uploaded to WhatsApp servers and then a new message is sent with the object id returned after upload the file.

Image Messages

Call the method **SendMessageImage** and pass the following parameters:

- **aTo:** phone number
- **aFileName:** full filename (with path) of the image file to send.
- **aFileType:**
  - image/jpeg
  - image/png
- **aCaption:** title of the image (optional).

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendFileImage("34605889421", "c:\images\image.png", "image/png");
```

Document Messages

Call the method **SendMessageDocument** and pass the following parameters:

- **aTo:** phone number
- **aFileName:** full filename (with path) of the document file to send.
- **aFileType:**
  - text/plain
  - application/pdf
  - application/vnd.ms-powerpoint
  - application/msword
  - application/vnd.ms-excel
  - application/vnd.openxmlformats-officedocument.wordprocessingml.document
  - application/vnd.openxmlformats-officedocument.presentationml.presentation
  - application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
- **aCaption:** title of the document (optional).

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendFileDocument("34605889421", "c:\MyDocuments\invoice.pdf", "application/pdf");
```

Audio Messages

Call the method **SendMessageAudio** and pass the following parameters:

- **aTo:** phone number

- **aFileName:** full filename (with path) of the audio file to send.
- **aFileType:**
  - audio/aac
  - audio/mp4
  - audio/mpeg
  - audio/amr
  - audio/ogg

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendFileAudio("34605889421", "c:\Music\audio.mp3", "audio/mp4");
```

Video Messages

Call the method **SendMessageVideo** and pass the following parameters:

- **aTo:** phone number
- **aFileName:** full filename (with path) of the video file to send.
- **aFileType:**
  - video/mp4
  - video/3gp

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendFileVideo("34605889421", "c:\Videos\video.mp4", "video/mp4");
```

Sticker Messages

Call the method **SendMessageSticker** and pass the following parameters:

- **aTo:** phone number
- **aFileName:** full filename (with path) of the sticker file to send.
- **aFileType:**
  - image/webp

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendFileSticker("34605889421", "c:\Stickers\MySicker.webp", "image/webp");
```

WhatsApp Download Media

If you receive a message with a media file link, you can download the media file using the method **DownloadMedia**.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->DownloadMedia("38923878928822", "c:\whatsapp\media\image.png");
```

To delete a previously uploaded media file, just call **DeleteMedia** and pass the object id as argument.

API Telegram

Telegram

Telegram offers two kinds of APIs, one is **Bot API** which allows you to create programs that use Bots and HTTPs as protocol. **Telegram API and TDLib** allow you to build customized Telegram clients and is much more powerful than Bot API.

sgcWebSockets **supports TDLib through tdjson** library, which means that you can build your own telegram client. TDLib takes care of all network implementation details, encryption and local data storage. TDLib supports all Telegram features.

TDLib (Telegram Database Library) Advantages

- **Cross-platform:** can be used on Windows, Android, iOS, MacOS, Linux...
- **Easy to use:** uses json messages to communicate between application and telegram.
- **High-performance:** In the Telegram Bot API, each TDLib instance handles more than 24000 bots.
- **Consistent:** TDLib guarantees that all updates will be delivered in the right order.
- **Reliable:** TDLib remains stable on slow and unreliable internet connections.
- **Secure:** All local data is encrypted using a user-provided encryption key.
- **Fully Asynchronous:** Requests to TDLib don't block each other. Responses will be sent when they are available.

Configuration

Windows

TDLib requires other third-parties libraries: OpenSSL and ZLib. These libraries must be deployed with tdjson library.

* Windows versions require VCRuntime, which can be downloaded from Microsoft: https://www.microsoft.com/en-us/download/details.aspx?id=52685, If after installing, the problem persist, try to copy the following dll in the same folder where your application is: VCRUNTIME140.dll and VCRUNTIME140_1.dll.

Copy the following libraries to the same directory where your application is located:

Windows 32	Windows 64
tdjson.dll	tdjson.dll
libcrypto-1_1.dll	libcrypto-1_1-x64.dll
libssl-1_1.dll	libssl-1_1-x64.dll
zlib1.dll	zlib1.dll

OSX64

Deploy the library libtdjson.dylib to your device and you can set where is the library using SetTDJsonPath, example:

if you deploy to "Contents\MacOS\", you must set the path in TPath.GetDirectoryName(ParamStr(0)) folder.

OSXARM64

Deploy the library libtdjson.dylib to your device and you can set where is the library using SetTDJsonPath, example:

if you deploy to "Contents\MacOS\", you must set the path in TPath.GetDirectoryName(ParamStr(0)) folder.

Linux64

Deploy the library libtdjson.so to your device and set the library path calling the method SetTDJsonPath.

Android

Deploy the library libtdjsonandroid.so to your device. Example: if you deploy an Android64 library, set RemotePath in Project/Deployment to "library\lib\arm64-v8a\". If is Android32, set RemotePath to "library\lib\armeabi-v7a\"

iOS64

Copy the library libtdjson.a to these directories:

- C:\Program Files (x86)\Embarcadero\Studio\<IDE Version>\lib\iosDevice64\debug
- C:\Program Files (x86)\Embarcadero\Studio\<IDE Version>\lib\iosDevice64\release

Creating your Telegram Application

In order to obtain an API id and develop your own application using the Telegram API you need to do the following:

- Sign up for Telegram using any application.
- Log in to your Telegram core: https://my.telegram.org.
- Go to **API development tools** and fill out the form.
- You will get basic addresses as well as the **api_id** and **api_hash** parameters required for user authorization.
- For the moment each number can only have one **api_id** connected to it.

These values must be set in **Telegram.API** property of Telegram component. In order to authenticate, you can authenticate as an user or as a bot, there are 2 properties which you can set to login to Telegram:

- **PhoneNumber:** if you login as an user, you must set your **phone number** (with international code), example: +34699123456
- **BotToken:** if you login as a bot, set your token in this property (as provided by telegram).
- **DatabaseDirectory:** allows you to specify where is the tdlib database. Leave empty and will take the default configuration.

The following parameters can be configured:

- **ApplicationVersion:** application version, example: 1.0
- **DeviceModel:** device model, example: desktop
- **LanguageCode:** user language code, example: en.
- **SystemVersion:** verison of operating system, example: windows.

Optionally, you can configure the path where is located tdjson library using **SetTDJsonPath** method. Just pass the path before start a new telegram session.

Once you have configured Telegram Component, you can set Active property to true and program will attempts to connect to Telegram.

Sample Code

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "your api hash";
oTelegram->Telegram->API->ApiId = "your api id";
oTelegram->PhoneNumber = "your phone number";
oTelegram->ApplicationVersion = "1.0";
oTelegram->DeviceModel = "Desktop";
oTelegram->LanguageCode = "en";
oTelegram->SystemVersion = "Windows";
oTelegram->Active = true;
```

Authorization

There are two events which can be called by library in order to get an Authentication Code (delivered in Telegram Application, not SMS) or to provide a password.

OnAuthenticationCode

This event is called when Telegram sends an Authorization Code to Telegram Application and user must copy this code and set in Code argument of this event.

```
void OnAuthenticationCode(TObject *Sender, ref string Code)
{
  Code = InputBox("Telegram Code", "Introduce code", "");
}
```

OnAuthenticationPassword

This event is called when Telegram requires that user set a password.

Authorization Status

Once authorization has started, you can check the status of authorization **OnAuthorizationStatus** event, this event is called every time there is a change in status of authorization. Some values of Status are:

- authorizationStateWaitTdlibParameters
- authorizationStateWaitEncryptionKey
- authorizationStateWaitPhoneNumber
- authorizationStateWaitCode
- authorizationStateLoggingOut
- authorizationStateClosed
- authorizationStateReady

Connection Status

Once connection has started, you can check the status of connection **OnConnectionStatus** event, this event is called every time there is a change in status of connection. Some values of Status are:

- connectionStateConnecting
- connectionStateUpdating
- connectionStateReady

Main Methods

TsgcTDLib_Telegram API Component support several Telegram methods, find below the most used.

Method	Parameters	Description
SendTextMessage	**aChatId:** Id of Chat which message will be sent **aText:** Text of Message. **InlineKeyboard:** Optional Buttons (only bots).	Sends a Text Message to a Chat
SendRichTextMessage	**aChatId:** Id of Chat which message will be sent **aText:** Text of Message. **InlineKeyboard:** Optional Buttons (only bots).	Sends a Rich Text Message to a Chat. Markdown syntax: • Bold: **bold**

		• Italic: __italic__ • Strike: --strike-- • Underline: ~~underline~~ • Code: ##code##
SendDocumentMessage	**aChatId:** Id of Chat which message will be sent **aFilePath:** full file path of document **aInlineKeyboard:** Optional Buttons (only bots).	Sends a Document to a Chat.
SendPhotoMessage	**aChatId:** Id of Chat which message will be sent **aFilePath:** full file path of photo **Width:** witdh of photo. **Height:** width of photo. **InlineKeyboard:** Optional Buttons (only bots).	Sends a Photo to a Chat.
SendVideoMessage	**aChatId:** Id of Chat which message will be sent **aFilePath:** full file path of video **aWidth:** witdh of video. **Height:** width of video. **aDuration:** duration of video in seconds. **aInlineKeyboard:** Optional Buttons (only bots).	Sends a Video to a Chat.
SendInvoiceMessage	**aChatId:** Id of Chat which message will be sent **aInvoice:** Text of Message. **aInlineKeyboard:** Optional Buttons (only bots).	Sends an Invoice (only available when is a Bot and in Private Channels).
EditTextMessage	**aChatId:** Id of Chat which message will be sent **aMessageId:** Id of Message to modify **Text:** Text of Message. **InlineKeyboard:** Optional Buttons (only bots). **ShowKeyboard:** Optional Buttons (only bots).	Edits the text of a message (or a text of a game message)
AddChatMember	**aChatId:** Id of Chat which message will be sent **aUserId:** Identifier of the user. **aForwardLimit:** The number of earlier messages from the chat to be forwarded to the new member; up to 100. Ignored for supergroups and channels.	Adds a new member to a chat. Members can't be added to private or secret chats. Members will not be added until the chat state has been synchronized with the server.
AddChatMembers	**aChatId:** Id of Chat which message will be sent **aUserIds:** Identifiers of the users to be added to the chat.	Adds multiple new members to a chat. Currently this option is only available for supergroups and channels. This option can't be used to join a chat. Members can't be added to a channel if it has more than 200 members. Members will not be added until the chat state has been synchronized with the server.
GetChatMember	**aChatId:** Chat Identifier. **aUserId:** User Identifier.	Returns information about a single member of a chat.
GetBasicGroupFullInfo	**aGroupId:** Basic Group Identifier	Returns full information about a basic group by its identifier.

GetSuper-groupMem-bers	**aSuperGroupId:** Identifier of the supergroup or channel. **aSupergroupMembersFilter:** The type of users to return. By default null **aOffset:** Number of users to skip. **aLimit:** The maximum number of users be returned; up to 200.	Returns information about members or banned users in a supergroup or channel.
JoinChatBy-InviteLink	**aLink:** Invite link to import;	Uses an invite link to add the current user to the chat if possible. The new member will not be added until the chat state has been synchronized with the server.
Create-NewSe-cretChat	**aUserId:** Identifier of the user.	Creates a new secret chat.
CreateNew-Basic-GroupChat	**aUserIds:** Identifiers of the users to be added to the chat. **aTitle:** Title of the new basic group	Creates a new basic group
CreateNew-Super-groupChat	**aTitle:** Title of the new SuperGroup **aIsChannel:** True, if a channel chat should be created. **aDescription:** Chat Description.	Creates a new supergroup or channel.
CreatePri-vateChat	**aUserId:** Identifier of the user. **aForce:** If true, the chat will be created without network request. In this case all information about the chat except its type, title and photo can be incorrect	Returns an existing chat corresponding to a given user
GetChats	**aOffsetOrder:** Chat order to return chats from **aOffsetChatId:** Chat identifier to return chats from **aLimit:** The maximum number of chats to be returned.	Returns an ordered list of chats. Chats are sorted by the pair (order, chat_id) in decreasing order (cannot be used is logged as Bot)
GetChat	**aChatId:** Chat identifier	Returns information about a chat by its identifier
GetChatHistory	**aChatId:** Chat identifier **aFromMessageId:** Identifier of the message starting from which history must be fetched; use 0 to get results from the last message. **aOffset:** Specify 0 to get results from exactly the from_message_id or a negative offset up to 99 to get additionally some newer messages. **aLimit:**The maximum number of messages to be returned	Returns messages in a chat. The messages are returned in a reverse chronological order
GetUser	**aUserId:** User Identifier	Returns information about a user by their identifier.
AddProxy-HTTP	**aServer:** Server name of proxy. **aPort:** Number of proxy port. **aUserName:** Username for logging in; may be empty. **aPassword:** Password for logging in; may be empty. **aHTTPOnly:** Pass true, if the proxy supports only HTTP requests and doesn't support transparent TCP connections via HTTP CONNECT method.	Adds a HTTP proxy server for network requests. Can be called before authorization.
AddProx-yMTProto	**aServer:** Server name of proxy.	Adds a MTProto proxy server for network requests. Can be called before authorization.

	aPort: Number of proxy port. **aSecret:** The proxy's secret in hexadecimal encoding.	
AddProxySocks5	**aServer:** Server name of proxy. **aPort:** Number of proxy port. **aUserName:** Username for logging in; may be empty. **aPassword:** Password for logging in; may be empty.	Adds a Socks5 proxy server for network requests. Can be called before authorization.
EnableProxy	**aId:** ID of proxy	Enables a proxy. Only one proxy can be enabled at a time. Can be called before authorization.
DisableProxy		Disables the currently enabled proxy. Can be called before authorization.
RemoveProxy	**aId:** ID of proxy	Removes a proxy server. Can be called before authorization.
GetProxies		Returns list of proxies that are currently set up. Can be called before authorization.
getChatSponsoredMessage	**aChatId:** ID of the chat	Returns sponsored message to be shown in a chat; for channel chats only. Returns a 404 error if there is no sponsored message in the chat.
ViewMessage	**aSponsorChatId:** ID of the sponsor Chat **aMessageId:** ID of the message	Informs TDLib that messages are being viewed by the user. Many useful activities depend on whether the messages are currently being viewed or not
Logout		Logouts from Telegram.
TDLibSend	**aRequest:** JSON Request.	Send any Request in JSON protocol.

Example How to send a Text Message

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "your api hash";
oTelegram->Telegram->API->ApiId = "your api id";
oTelegram->PhoneNumber = "your phone number";
oTelegram->Active = true;
...
oTelegram->SendTextMessage("1234", "My First Message from sgcWebSockets");
```

Example How to send a method not implemented

You can Send Any JSON message using TDLibSend method, example: join a telegram chat.

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "your api hash";
oTelegram->Telegram->API->ApiId = "your api id";
oTelegram->PhoneNumber = "your phone number";
oTelegram->Active = true;
...
oTelegram->TDLibSend("{\"@type\": \"joinChat\", \"chat_id\": \"1234\"}");
```

Events

OnBeforeReadEvent
This event is called when JSON message is received by Telegram API component and is still not processed. Set Handled property to True if you process this event manually or don't want that event is processed by component. You can use this event to log all messages too.

OnMessageText
This event is called when a New Message Text has been received, read MessageText parameter to access to message text properties.

- **ChatId:** Chat Identifier.
- **MessageId:** Message Identifier.
- **SenderUserId:** Sender Identifier.
- **Text:** Text of message.

OnMessageDocument
This event is called when a New Document Message is received. Access to MessageDocument to get access to Document properties.

- **ChatId:** Chat Identifier.
- **MessageId:** Message Identifier.
- **SenderUserId:** Sender Identifier (read SenderChat and SenderUser from tdlib 1.7.+).
- **FileName:** Name of Document.
- **DocumentId:** Document Identifier.
- **LocalPath:** full path to local file if exists.
- **MimeType:** Mime-type of document.
- **Size:** Size of Document.
- **RemoteDocumentId:** Remote Document Identifier.

OnMessagePhoto
This event is called when a New Photo Message is received. Access to MessagePhoto to get access to Photo properties.

- **ChatId:** Chat Identifier.
- **MessageId:** Message Identifier.
- **SenderUserId:** Sender Identifier (read SenderChat and SenderUser from tdlib 1.7.+).
- **PhotoId:** Photo Identifier.
- **LocalPath:** full path to local file if exists.
- **Size:** Size of Photo.
- **RemotePhotoId:** Remote Photo Identifier.

OnVideoPhoto
This event is called when a New Video Message is received. Access to MessageVideo to get access to Video properties.

- **ChatId:** Chat Identifier.
- **MessageId:** Message Identifier.
- **SenderUserId:** Sender Identifier (read SenderChat and SenderUser from tdlib 1.7.+).
- **VideoId:** Photo Identifier.
- **LocalPath:** full path to local file if exists.
- **Width:** width of video.
- **Height:** height of video.
- **Duration:** duration in seconds of video.
- **Size:** Size of Video.
- **RemoteVideoId:** Remote Photo Identifier.

OnMessageSponsored
This event is called when a New Sponsored Message has been received (after calling the method getChatSponsoredMessage)

- **SponsorChatId:** Sponsor Chat Identifier.
- **MessageId:** Message Identifier.
- **Text:** Text of message.

OnNewChat
This event is called when a new chat is received.

- **ChatId:** Chat Identifier.
- **ChatType:** Chat Type (chatTypeSupergroup, chatTypePrivate...)
- **Title:** Chat name.
- **SuperGroupId:** Group Id if is a SuperGroup.
- **IsChannel:** returns if is channel or not.

OnNewCallbackQuery
This event is called when a new incoming callback query is received; for bots only.

- **Id:** Unique query identifier.
- **SenderUserId:** Identifier of the user who sent the query.
- **ChatId:** Identifier of the chat, in which que query was sent.
- **MessageId:** Identifier of the message, from which the query originated.
- **ChatInstance:** Identifier that uniquely corresponds to the chat to which the message was sent.
- **PayloadData:** the payload from a general callback button.
    - **Data:** Data that was attached to the callback button.

OnEvent
This event is called when a new Event is received by API Component. Can be used to process some events not implemented by API Component.

- **Event:** Event name (events like: updateOption, updateUser...)
- **Text:** full JSON message

OnException
This event is called if there is any exception when processing Telegram API Data.

Properties

MyId: returns the User Identifier of current user.

Full Code Sample

Check the following code sample which shows how connect to Telegram API, ask user to introduce a Code (if required by Telegram API), send a message when connection is ready and Log Text Messages received.

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "your api hash";
oTelegram->Telegram->API->ApiId = "your api id";
oTelegram->PhoneNumber = "your phone number";
oTelegram->ApplicationVersion = "1.0";
oTelegram->DeviceModel = "Desktop";
oTelegram->LanguageCode = "en";
oTelegram->SystemVersion = "Windows";
oTelegram->Active = true;

void OnAuthenticationCode(TObject *Sender, ref string Code)
{
  Code = InputBox("Telegram Code", "Introduce code", "");
}

void OnMessageText(TObject *Sender, TsgcTelegramMessageText *MessageText)
{
  Log("Message Received: " + MessageText->Text);
}

void OnConnectionStatus(TObject *Sender, const string Status)
{
  if (Status == "connectionStateReady")
  {
    oTelegram->SendTextMessage("1234", "Hello Telegram!");
  }
}
```

Telegram | Send Telegram Message With Inline Buttons

Telegram API allows you to send messages with inline buttons to select options as an answer (this option is only available for bots).

Before you send a message create an instance of the class **TsgcTelegramReplyMarkupInlineKeyboard** and call the method **AddButtonTypeCallback** or **AddButtonTypeUrl** for every button you want to create.

Example

Create a new message asking the user if likes or not the message and a link to answer a poll. Process the response using **OnNewCallbackQuery** event.

```
TsgcTelegramReplyMarkupInlineKeyboard *oReplyMarkup = new TsgcTelegramReplyMarkupInlineKeyboard();
try
{
  oReplyMarkup->AddButtonTypeCallback("Yes", "I like it");
  oReplyMarkup->AddButtonTypeCallback("No", "I hate it");
  oReplyMarkup->AddButtonTypeUrl("Poll", "https://www.yoursite.com/telegram/poll");
  sgcTelegram->SendTextMessage("123456", "Do you like the message?", oReplyMarkup);
}
__finally
{
  oReplyMarkup->Free();
}

void OnNewCallbackQuery(TObject *Sender, TsgcTelegramCallbackQuery *CallbackQuery)
{
  if (CallbackQuery->PayloadData->Data == "I like it") then
  {
    ShowMessage("yes")
  }
  else
  {
    ShowMessage("no");
  }
}
```

Telegram | Send Bot Message With Buttons

Telegram API allows you to send messages with buttons to request data from the user (this option is only available for bots).

Before you send a message create an instance of the class **TsgcTelegramReplyMarkupShowKeyboard** and call the method **AddButtonTypeRequestLocation, AddButtonTypeRequestPhoneNumber** or **AddButtonTypeText** for every button you want to create.

Example

Create a new message asking the user to provide the PhoneNumber

```
oReplyMarkup = new TsgcTelegramReplyMarkupShowKeyboard();
oReplyMarkup->AddButtonTypeRequestPhoneNumber("Give me your phone");
sgcTelegram->SendTextMessage("123456", "Please provide the information below", null, oReplyMarkup);
oReplyMarkup->Free();
```

Telegram | Send Telegram Message Bold

You can highlight text messages using bold, italic and more styles. Use the method **SendRichTextMessage**, to send a Text message with style capabilities, this method parses the text message and adds the entities required automatically to the API Telegram.

Markdown Syntax

- Bold [*]

```
**This is Bold**
```

- Italic [_]

```
__This is Italic__
```

- Strike [-]

```
--This is Strike--
```

- Underline [~]

```
~~This is Underline~~
```

- Code [#]

```
##This is Monospace##
```

Telegram | Chat not found as Bot

When you **log as bot**, the GetChats method cannot be used, so you don't get All available chats. If it's the **first time you login as Bot** and you try to **send a message** to a **known Chat**, you will get this **error**:

```
{"@type":"error","code":5,"message":"Chat not found"}
```

The solution is to call the **GetChat** method before sending a telegram message and pass the **ChatId** as a parameter. Once you get the Chat data, you can send telegram messages as usual.

As a note, you **only** need to **call GetChat** the **FIRST TIME** before sending a message if you have never received any bot message from this chat. If you close the application and start again, there is no need to call GetChat first because the Chat is already saved in the telegram database.

Telegram | Sponsored Messages

Each time the user opens a channel, channels.getSponsoredMessages must be called to receive sponsored messages available for this channel. The result must be cached for 5 minutes.

Displaying sponsored messages

Sponsored messages must be displayed below all other posts in the channel, after the user scrolls further down, past the last message. The promoted channel or bot specified in the from_id field must be displayed as the author of the message. The message should also contain one of the following buttons at the bottom:

- **View Bot:** if a bot is being promoted. Tapping the button must open the chat with the bot. If start_param is specified, the app must use the deep linking mechanism to open the bot.
- **View Channel:** if a channel is being promoted. Tapping the button must open the channel.
- **View Post:** if a channel is being promoted and channel_post is specified. Tapping the button must open the particular channel post.

Once the entire text is shown on the screen (excluding the button), **ViewMessage** method must be called with the random_id of this sponsored message.

Get Sponsored Messages

Send a request to the channel asking if there are sponsored messages available, just call the method **GetChatSponsoredMessage**.

```
TsgcTDLib_Telegram *oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "ABCDEFGHIJKLMN";
oTelegram->Telegram->API->ApiId = "1234";
oTelegram->PhoneNumber = "008745744155";
oTelegram->Active = true;
oTelegram->getChatSponsoredMessage("100");
```

If the chat has sponsored messages, the event **OnMessageSponsored** is called with the content of the Sponsored message. If there are no messages, a 404 error is returned.

```
private void(TObject *Sender, TsgcTelegramMessageSponsored *MessageSponsored)
{
  DoLog(MessageSponsored->Text);
}
```

Call the method **ViewMethod** after the Sponsored Messages has been shown to the user.

```
oTelegram->ViewMessage("100", "54653256245");
```

Telegram | Send Telegram Invoice Message

If your bot supports inline mode, users can also send invoices to other chats via the bot, including to one-on-one chats with other users.

Invoice messages feature a photo and description of the product along with a prominent Pay button. Tapping this button opens a special payment interface in the Telegram app

The bots can send invoices as a message using the method **SendInvoiceMessage**.

```
private void SendInvoice()
{
  TsgcTelegramSendInvoice *oInvoice = new TsgcTelegramSendInvoice();
  Try
  {
    oInvoice->Title = "Invoice Title Test";
    oInvoice->Description = "Description Invoice Test";
    oInvoice->Invoice->Currency = 'EUR';
    oInvoice->Invoice->Total = 800;
    oInvoice->Invoice->IsTest = True;
        oInvoice->Invoice->Payload := "payload";
        oInvoice->Invoice->ProviderToken := "provider_token";
        oInvoice->Invoice->ProviderData := "provider_data";

    sgcTelegram->SendInvoiceMessage("3284239872", oInvoice);
  __finally
  {
    oInvoice->Free();
  }
}
```

Telegram | Get SuperGroup Members

Telegram API allows you to get information about members of a SuperGroup. Use the method **GetSuperGroup-Members** to get information about members or banned users in a supergroup or channel. Can be used only if SupergroupFullInfo.can_get_members is true; additionally, administrator privileges may be required for some filters.

By default the method returns All members of the group, but you can filter the members returned using the Filter parameter. There are the following parameters:

tsgmFilterNone
Default value, means members are not filtered.

tsgmFilterAdministrators
Returns the creator and administrators.

tsgmFilterBanned
Returns users banned from the supergroup or channel; can be used only by administrators.
You can use the argument aSuperGroupMembersQuery to search using a query.

tsgmFilterBots
Returns bot members of the supergroup or channel.

tsgmFilterContacts
Returns contacts of the user, which are members of the supergroup or channel.
You can use the argument aSuperGroupMembersQuery to search using a query.

tsgmFilterMention
Returns users which can be mentioned in the supergroup.

tsgmFilterRecent
Returns recently active users in reverse chronological order.

tsgmFilterRestricted
Returns restricted supergroup members; can be used only by administrators.
You can use the argument aSuperGroupMembersQuery to search using a query.

tsgmFilterSearch
Used to search for supergroup or channel members via a (string) query.
You can use the argument aSuperGroupMembersQuery to search using a query.

You can read the result of the result using OnEvent callback and filtering by event = "chatMembers".

```
Telegram->GetSupergroupMembers(1452979380);


private void OnTelegramEvent(TObject *Sender, const string Event, const string Text)
{
  if (Event == "chatMembers")
  {
    ReadJSON(Text);
  }
}
```

Telegram | Add Telegram Proxy

Telegram Client can be configured to make use of a proxy. Currently, Telegram supports 3 types of proxies:

1. HTTP
2. MTProto
3. Socks5

Add Proxy

In order to configure a HTTP Proxy, first you must add the proxy to telegram configuration, to do this, just call **AddProxyHTTP** and if successful, a message will be returned with the new proxy added. Once the proxy has been added to the list, just call **EnableProxy** and pass the **ID of the proxy** received on the confirmation message.

```
Telegram->AddProxyHTTP("8.8.8.8", 8080, "", "", true);
// ... read the confirmation message and save the ID of the proxy.
Telegram->EnableProxy(2);
```

Remove Proxy

Call **RemoveProxy** method and pass the ID of the proxy you want to remove.

Telegram | Register Telegram User

The process to register a new user in Telegram is very simple, you need your API Id and API Hash, and the phone number of the new account.

Configure the telegram client:

- API Id
- API Hash
- Telephone Number of the new telegram account.

Start the client and a new code will be sent to the phone, the client will ask for the telegram code and if it's correct, the event OnRegisterUser will be called. In this event set the First Name and Last Name of the user and the registration will be completed.

```
TsgcTDLib_Telegram *oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "ABCDEFGHIJKLMN";
oTelegram->Telegram->API->ApiId = "1234";
oTelegram->PhoneNumber = "008745744155";
oTelegram->Active = true;

void OnTelegramAuthenticationCode(TObject *Sender, ref string Code)
{
  Code = "code sent to phone";
}

void OnTelegramRegisterUser(TObject *Sender, ref string FirstName, ref string LastName)
{
  FirstName = "first name";
  LastName = "last name";
}
```

RCON

RCON

The Source RCON Protocol is a TCP/IP-based communication protocol used by Source Dedicated Server, which allows console commands to be issued to the server via a "remote console", or RCON. The most common use of RCON is to allow server owners to control their game servers without direct access to the machine the server is running on.

Configuration

The **RCON_Options** allows you to configure the following properties:

- **Host:** server remote address.
- **Port:** server listening port.
- **Password:** is the secret string used to authenticate against the server

Connect

Use the property **Active** to Connect / Disconnect from server.
When Active is set to True, the client tries to connect to the server, if it can connect, it will try to authenticate using the provided password.
The server will send a response to an authentication request. The event **OnAuthenticate** will be called and you can read if authentication is successful or not using the Authenticate parameter.

Send Commands

Use the method **ExecCommand** to send commands to the server. The responses will be available **OnResponse** Event.

```
TsgcLib_RCON oRCON = new TsgcLib_RCON();
oRCON->RCON_Options->Host = "127.0.0.1";
oRCON->RCON_Options->Port = 25575;
oRCON->RCON_Options->Password = "test";
oRCON->Active = true;


void OnAuthenticate(TObject *Sender, bool Authenticated, const TsgcRCON_Packet *aPacket)
{
  if (Authenticated == true)
  {
    DoLog("#authenticated");
  }
  else
  {
    DoLog("#not authenticated");
  }
}


void OnResponse(Object *Sender, const string aResponse, const TsgcRCON_Packet *aPacket)
{
  DoLog(aResponse);
}
```

CryptoHopper

CryptoHopper

CryptoHopper is an automated crypto trading bot that allows you to automate trading and portfolio management for Bitcoin, Ethereum, Litecoin and more.

Configuration

Requires a **Developer Account** and once you have been approved you can start to create a new App. The API uses OAuth2 to authenticate, so you can retrieve the **client_id** and **client_secret** from your App.

```
TsgcHTTP_Cryptohopper oCryptoHopper = new TsgcHTTP_Cryptohopper();
oCryptoHopper->CryptoHopperOptions->OAuth2->ClientId = "client_id";
oCryptoHopper->CryptoHopperOptions->OAuth2->ClientSecret = "client_secret";
oCryptoHopper->CryptoHopperOptions->OAuth2->LocalIP = "127.0.0.1";
oCryptoHopper->CryptoHopperOptions->OAuth2->LocalPort = 8080;
oCryptoHopper->CryptoHopperOptions->OAuth2->Scope->Text = "read,notifications,manage,trade";
```

Methods

CryptoHopper uses HTTPs as the protocol to send Requests to the API. Some methods require authentication (place orders, retrieve user data...) and some others are public (get exchange data for example).

The functions return the CryptoHopper response and if there is any error an exception will be raised.

Hoppers

Manage Basic Hopper Operations.

Method	Arguments	Description
GetHoppers		Get Hoppers of users.
CreateHopper	**aBody:** configuration json text.	Create a new Hopper.
GetHopper	**aId:** hopper id	Retrieve Hopper
DeleteHopper	**aId:** hopper id	Delete Hopper
UpdateHopper	**aId:** hopper id **aBody:** configuration json text.	Update Hopper

Orders

Manage the Orders of your Hopper.

Method	Arguments	Description
GetOpenOrders	**aId:** hopper id	Retrieve all of the open orders of the hopper.
CreateNewOrder	**aId:** hopper id **aOrder:** instance of TsgcHTTPCTHOrder	Create new buy or sell order. For sell, rather use the sell endpoint.
PlaceMarketOrder	**aId:** hopper id **aOrderSide:** cthosBuy or cthosSell. **aCoin:** coin name, example: EOS **aAmount:** order size.	Place a Market Order.
PlaceLimitOrder	**aId:** hopper id **aOrderSide:** cthosBuy or cthosSell. **aCoin:** coin name, example: EOS **aAmount:** order size. **aPrice:** limit price.	Place a Limit Order
DeleteOrder	**aId:** hopper id **aOrderId:** order id	Deletes order for selected hopper.
DeleteAllOrders	**aId:** hopper id	Deletes all open order for selected hopper.
GetOpenOrder	**aId:** hopper id **aOrderId:** order id	Get open order in hopper by id.
CancelOrder	**aId:** hopper id **aOrderId:** order id	Cancel an open order.

Position

Manage the Positions of your Hopper.

Method	Arguments	Description
GetPosition	**aId:** hopper id	Get open positions of hopper.

Trade

Trade History from your Hopper.

Method	Arguments	Description
GetTradeHistory		Get the trade history of the hopper.
GetTradeHistoryById	**aId:** hopper id **aTradeId:** trade id	Get a trade by id of the hopper.

Exchange

Get Information from available exchanges on CryptoHopper

Method	Arguments	Description
GetExchange		Get all available exchanges on Cryptohopper.
GetAllTickers	**aExchange:** exchange name	Get ticker for all pairs
GetMarketTicker	**aExchange:** exchange name **aPair:** pair name	Get ticker from market pair.
GetOrderBook	**aExchange:** exchange name **aPair:** pair name	Gets the orderbook for the selected exchange, market and orderbook depth.

	aDepth: order book depth	

Webhooks

Trade History from your Hopper.

Method	Arguments	Description
CreateWebhook	**aURL:** webhook url **aMessageTypes:** message types separated by comma.	Update or create a Webhook
DeleteWebhook	**aURL:** webhook url	Delete an existing Webhook.

Signals

Send Signals to CryptoHopper API.

Method	Arguments	Description
SendSignal	**aSignal:** is the class with all the fields required to send a signal.	Sends a Signal
SendTestSignal	**aSignal:** is the class with all the fields required to send a signal.	Sends a Test Signal
GetSignalStats	**aSignalId:** id of the signal. **aExchange:** optional, name of the exchange.	Retrieve some of the signal statistics.

How to Update Cryptohopper Config

Use the UpdateHopper method to update the Hopper Configuration. The method is overloaded so you can pass the JSON string or use the object TsgcHTTPCTHopper and use the properties to enable or disable the Hopper Properties.

```
public string EnableHopper()
{
  TsgcHTTPCTHopper *oHopper = new TsgcHTTPCTHopper();
  try
  {
    oHopper->Enabled = 1;
    result = Cryptohopper->UpdateHopper("1234", oHopper);
  }
  __finally
  {
    oHopper->Free();
  }
}
```

How to Configure Webhook

Webhook allows you to receive notifications when something happens in a hopper. Webhooks require a public HTTPs Server which will listen in a URL address all messages sent by cryptohopper. The public server needs to be protected with a SSL certificate (self-signed certificates are not allowed).

First you must create a webhook, so configure the Webhook property of Cryptohopper client setting the Host and Port when the server will be listening. Then configure the certificate in SSLOptions property.

Example: The public IP address will be 1.1.1.1 and the listening port will be 443. The certificate is stored as PEM file with sgc.pem filename and without password.

```
/* OAuth2 */
cryptohopper->CryptohopperOptions->OAuth2->ClientId = "client_id";
cryptohopper->CryptohopperOptions->OAuth2->ClientSecret = "client_secret";
cryptohopper->CryptohopperOptions->OAuth2->LocalIP = "127->0->0->1";
cryptohopper->CryptohopperOptions->OAuth2->LocalPort = 8080;
/* Webhook */
cryptohopper->CryptohopperOptions->Webhook->Enabled = True;
cryptohopper->CryptohopperOptions->Webhook->Host = "1.1.1.1";
cryptohopper->CryptohopperOptions->Webhook->Port = 443;
cryptohopper->CryptohopperOptions->Webhook->ValidationCode = "1234";
cryptohopper->CryptohopperOptions->Webhook->SSLOptions->CertFile = "sgc->pem";
cryptohopper->CryptohopperOptions->Webhook->SSLOptions->KeyFile = "sgc->pem";
cryptohopper->CryptohopperOptions->Webhook->SSLOptions->RootCertFile = "sgc->pem";
cryptohopper->CryptohopperOptions->Webhook->SSLOptions->Password = "";
cryptohopper->StartWebhook();
```

WebPush

RFC 8030
RFC 8291

The WebPush protocol is defined by the **RFC 8030** (Delivery using HTTP Push) and **RFC 8291** (Message Encryption).

Web Push is a **standardized protocol** for **delivering push notifications to web browsers**. It uses the Push API, which is a standard web API that enables websites to register and receive push messages. The Push API allows a website to send push messages to a user's browser, even when the user is not actively browsing the website.

To use Web Push, a website first needs to **obtain a push subscription from the user's browser**. The subscription consists of a unique endpoint URL and an encryption key. The endpoint URL is a URL that the website can use to send push messages to the user's browser, and the encryption key is used to encrypt and decrypt the push messages.

Once the website has **obtained a push subscription**, it can **send push messages** to the user's browser by making an HTTP request to the endpoint URL. The push message is sent in a special format called the Web Push Protocol Message, which consists of a set of headers and a payload. The headers contain information such as the encryption key and the TTL (time-to-live) of the message, while the payload contains the actual content of the message.

When the **user's browser receives a push message**, it **first decrypts the message** using the encryption key. It then **displays the notification to the user**, along with any additional actions that the user can take, such as dismissing the notification or opening the website.

To ensure the security and privacy of push messages, Web Push uses end-to-end encryption and requires that push subscriptions be obtained over a secure connection (e.g., HTTPS). Additionally, the protocol provides mechanisms for authenticating the sender of a push message and preventing abuse (e.g., by limiting the number of push messages that a website can send to a user).

Components

There are 2 components which support WebPush:

- **TsgcWSAPIServer_WebPush:** implements the WebPush protocol on the server side, allowing you to ask users for permission, register subscriptions, send notifications, and more. This component already encapsulates a WebPush client to send notifications.
- **TsgcWebPush_Client:** implements the WebPush protocol on the client side, allowing you to send notifications to users via desktop and mobile web. This is useful if you already have the keys and endpoint and only want to publish WebPush messages to the subscribed clients.

TsgcWSAPIServer_WebPush

TsgcWSAPIServer_WebPush component.

Introduction

TsgcWSServer_API_WebPush is a component that provides functionality for handling WebPush subscriptions. WebPush is a protocol for delivering real-time notifications to web applications that run in the browser. This component can be used to manage subscriptions and send notifications to subscribed clients. Find below the properties, events, and methods provided by TsgcWSServer_API_WebPush class, along with code examples that demonstrate how to use them.

Configuration

1. **Attach a TsgcWSServer_API_WebPush** to a WebSocket server using the Server property.
2. Configure the **public and private keys** in the **WebPush.VAPID** property. (Registered users can download an executable that generates the VAPID keys for windows).
3. Requires deploying the **OpenSSL 3.0.0 version**
4. In the **WebPush.Endpoints** property you can define your own endpoints to handle the WebPush subscriptions; by default, accessing the "/sgcWebPush.html" endpoint will show a simple webpage that allows you to subscribe to the WebPush notifications.
5. Start the server and access the endpoint configured to test it.

Reference

- Properties — 3 properties available on the component.
- Methods — 3 public methods.
- Events — 3 events you can handle.

TsgcWebPush_Client

The TsgcWebPush_Client is a class that allows you to send a notification once you obtain the subscription details.

Introduction

The TsgcWebPush_Client is a class that allows you to send a notification once you obtain the subscription details.

Find below an example of using the WebPush client to send a notification given an endpoint, public key and authentication secret from a WebPush subscription.

```
void SendWebPushNotification()
{
    TsgcHTTP_API_WebPush_PushSubscription* oSubscription = new TsgcHTTP_API_WebPush_PushSubscription();
    try
    {
        oSubscription->Endpoint = "endpoint";
        oSubscription->PublicKey = "public key";
        oSubscription->AuthSecret = "authentication secret";
        TsgcHTTP_API_WebPush_Client* oWebPush = new TsgcHTTP_API_WebPush_Client(NULL);
        try
        {
            oWebPush->VAPID->PEM->PrivateKey->Text = "private_key_pem";
            oWebPush->VAPID->DER->PrivateKey = "private_key";
            oWebPush->VAPID->DER->PublicKey = "public_key";
            oWebPush->SendNotification(oSubscription, "{\"title\": \"eSeGeCe Notification\", \"body\": \"Hell
        }
        __finally
        {
            oWebPush->Free();
        }
    }
    __finally
    {
        oSubscription->Free();
    }
}
```

Reference

- Properties — 3 properties available on the component.
- Methods — 1 public methods.
- Events — 0 events you can handle.

Extensions

WebSocket protocol is designed to be extended. WebSocket Clients may request extensions and WebSocket Servers may accept some or all extensions requested by clients.

Extensions supported:

1. Deflate-Frame: compress WebSocket frames.

2. PerMessage-Deflate: compress WebSocket messages.

Extensions | PerMessage-Deflate

PerMessage is a WebSocket protocol extension, if the extension is supported by Server and Client, both can compress transmitted messages:

- Uses Deflate as the compression method.
- Compression only applies to Application data (control frames and headers are not affected).
- Server and client can select which messages will be compressed.

Max Window Bits

This extension allows customizing the server and client size of the sliding window used by the LZ77 algorithm (between 8 and 15). The greater this value, the more likely it is to find and eliminate duplicates, but it consumes more memory and CPU cycles. 15 is the default value.

No Context Take Over

By default, previous messages are used for compression and decompression. If messages are similar, this improves the compression ratio. If enabled, then each message is compressed using only its own message data. By default, it is disabled.

MemLevel

This value is not negotiated between server and client. When set to 1, it uses the least memory, but slows down the compression algorithm and reduces the compression ratio; when set to 9, it uses the most memory and delivers the best performance. By default, it is set to 1.

** Indy version provided with Rad Studio XE2 raises an exception because of zlib version mismatch with initialization functions. To fix this, just update your Indy version to the latest.*

Extensions | Deflate-Frame

This is a WebSocket protocol extension that allows the compression of frames sent using the WebSocket protocol, supported by WebKit browsers like Chrome or Safari. This extension is supported on Server and Client components.

This extension has been deprecated.

** Indy version provided with Rad Studio XE2 raises an exception because of zlib version mismatch with initialization functions. To fix this, just update your Indy version to the latest.*

MCP

MCP (Model Context Protocol) is a standardized protocol designed to define and manage the **contextual exchange of information** between language models and external systems.

It allows structured, secure, and dynamic communication that enables AI models to **understand**, **extend**, and **interact** with external tools or data sources beyond their native environment.

MCP provides a **uniform interface** for passing contextual metadata, enabling models to interpret instructions, retrieve data, and execute tasks in a predictable and interoperable way similar in spirit to how *WebAuthn* standardizes authentication flows.

Purpose

MCP was developed to solve the challenges of **context fragmentation** and **integration inconsistency** between models, clients, and tools.
With MCP, any compliant client can communicate with any MCP-compatible model or service through a shared, well-defined structure.

Typical use cases include:
- Attaching **context metadata** (user session, app state, permissions, etc.) to model queries.
- Exchanging **function or tool definitions** dynamically.
- Managing **stateful interactions** across model sessions.
- Establishing **secure interoperability** between LLMs, SDKs, and backend services.

Protocol versions

The sgcWebSockets MCP components support the MCP protocol versions **2026-07-28**, **2025-11-25**, **2025-06-18** and **2025-03-26**.

The server works in a dual-era way, with no configuration needed:
- **2025-11-25 and older clients** start with `initialize` and use a session (`Mcp-Session-Id`) on every request, exactly as before.
- **2026-07-28 clients** are served statelessly. There is no `initialize` and no session; every request carries its protocol version, client information and capabilities in `_meta`, and the client can learn the server features with `server/discover`.

Server features of the protocol version 2026-07-28:
- Protocol Versions: era detection, `server/discover`, HTTP headers validation, error codes, cache hints, instructions, extensions and `x-mcp-header`.
- Subscriptions: notifications delivered through `subscriptions/listen` streams.
- Multi Round-Trip Requests: ask the client for elicitation, sampling or roots with `input_required` results.
- Tasks: run long tool calls in the background with the `io.modelcontextprotocol/tasks` extension.
- Logging and Progress: log and progress notifications streamed in the response of each request.

Client features of the protocol version 2026-07-28:
- Protocol Versions: automatic era detection with fallback to legacy servers, `server/discover` and `x-mcp-header`.
- Multi Round-Trip Requests: answer `input_required` results automatically or with **OnMCPInputRequired**.
- Subscriptions: receive list changes and resource updates with `subscriptions/listen`.
- Tasks: follow long tool calls that run as tasks on the server.
- Authorization: discovery, registration and login against the authorization server of a protected MCP server.

Components

- **TsgcWSAPIServer_MCP**: The component provides a simple but powerful solution for implementing the MCP server, with features like Tools, Prompts and Resources requests, Authentication, Flow control using events and much more.
- **TsgcWSAPIClient_MCP**: The component provides a simple but powerful solution for implementing the MCP client, with features like Tools, Prompts and Resources requests and more.

TsgcWSAPIServer_MCP

TsgcWSAPIServer_MCP component.

Introduction

TsgcWSAPIServer_MCP exposes the Model Context Protocol (MCP) over an sgcWebSockets HTTP server endpoint. The component bridges incoming HTTP requests with the TsgcAI_MCP_Server engine so that MCP-compatible clients can negotiate sessions, enumerate prompts/resources/tools, and invoke tools through JSON-RPC style calls.

Configuration

- **Place components:** The TsgcWSAPIServer_MCP component must be attached to an HTTP server, TsgcWebSocketHTTP_Server or TsgcWebSocketServer_HTTPAPI using the Server property.
- **Pick an endpoint:** Adjust EndpointOptions.Endpoint if the default /mcp path should change. Requests whose document equals this path are treated as MCP calls; others fall back to the parent API's resource handling.
- **Configure MCP settings:** Use **MCPOptions.ServerInfo** to name/version/title/icon the server, **MCPOptions.SessionTimeout** to control session lifetime, **MCPOptions.TasksSupported** to advertise the MCP tasks capability, and **MCPOptions.AuthenticationOptions** to require an API key or a custom HTTP header before any JSON-RPC call is processed.
- **Register prompts, tools, resources and resource templates:** Populate the read-only **Tools**, **Prompts**, **Resources** and **ResourceTemplates** lists exposed by the component.
- **Handle events:** Wire the published events described below to plug in business logic for session lifecycle, prompt/resource resolution, or tool execution. Exceptions can be intercepted to customize HTTP response codes.

Built-in HTTP endpoints (no configuration required)

When the component is attached to an HTTP server, the following endpoints and behaviors are served automatically alongside `/mcp`, without any new property to enable them. This enables out-of-the-box compatibility with web-based MCP connectors such as **claude.ai**, which require OAuth 2.1 with Dynamic Client Registration over CORS.

- **CORS support with origin reflection.** An `OPTIONS` preflight to any handled path returns `204 No Content` with the following headers:
  - `Access-Control-Allow-Origin`: echo of the request `Origin` header, or `*` if missing.
  - `Access-Control-Allow-Methods: POST, GET, DELETE, OPTIONS`
  - `Access-Control-Allow-Headers: content-type, mcp-session-id, mcp-protocol-version, authorization, accept`
  - `Access-Control-Expose-Headers: mcp-session-id`
  - `Access-Control-Max-Age: 86400`
  - `Strict-Transport-Security: max-age=31536000`

  All non-OPTIONS responses also include `Access-Control-Allow-Origin`, `Access-Control-Expose-Headers` and `Strict-Transport-Security`.

- **401 Unauthorized with `WWW-Authenticate`.** When `MCPOptions.AuthenticationOptions.ApiKey` (or `CustomHeader`) is enabled and the request has no or wrong credentials, the server responds with `401` and a header of the form:

  ```
  Bearer realm="MCP", resource_metadata="https://<host>/.well-known/oauth-protected-resource"
  ```

- **OAuth 2.1 discovery and flow endpoints.** The following endpoints are served on the same HTTP server as `/mcp`, supporting PKCE (S256), authorization-code and refresh-token grants, and Dynamic Client Registration:

Path	Spec	Behavior
`GET /.well-known/oauth-authorization-server`	RFC 8414	JSON authorization server metadata. Advertises PKCE (S256), `authorization_code` + `refresh_token` grants, public-client + `client_secret_basic` auth methods, dynamic registration at `/oauth/register`, and the `mcp` scope.
`GET /.well-known/oauth-protected-resource`	RFC 9728	JSON describing the MCP endpoint as a protected resource and pointing at this server as the authorization server.
`POST /oauth/register`	RFC 7591 (DCR)	Accepts a JSON client registration body (`client_name`, `redirect_uris`). Issues a generated `client_id` (no `client_secret` — public client). Returns `201` with the registered client information response.
`GET /oauth/authorize`	RFC 6749 + RFC 7636	Renders an HTML consent page. Required query params: `client_id`, `redirect_uri`, `response_type=code`, `code_challenge`, `code_challenge_method=S256`, `state`, `scope`.
`POST /oauth/authorize`	RFC 6749	Processes the consent form. The user must enter the **same value as** `MCPOptions.AuthenticationOptions.ApiKey.Value` to approve. On approve, redirects to `redirect_uri` with `?code=...&state=...`. On deny or wrong key, redirects with `?error=access_denied&state=...` or re-renders the form.
`POST /oauth/token`	RFC 6749 + RFC 7636	Form-encoded body. Supports `grant_type=authorization_code` (validates PKCE: `BASE64URL(SHA256(code_verifier)) == code_challenge`) and `grant_type=refresh_token`. Issues a 1-hour bearer access token plus a refresh token. Auth codes expire after 60 seconds and are single-use.

- **Bearer token validation extended.** When `MCPOptions.AuthenticationOptions.ApiKey.Enabled` is `True`, the `Authorization: Bearer <X>` header on `/mcp` requests is accepted if `<X>` matches **either** the configured static `ApiKey.Value`, **or** a currently-valid OAuth-issued access token (issued via the flow above).
- **In-memory storage.** OAuth clients, codes and tokens are kept in memory only. They are lost on server restart.
- **Override via `OnMCPHTTPRequest`.** Handlers wired to `OnMCPHTTPRequest` can short-circuit any of the above behaviors by setting `Handled := True`.

Running over stdio (TsgcAI_MCP_Server_Stdio)

Besides the HTTP transports, the MCP server engine can also run over the **stdio** transport. The **TsgcAI_MCP_Server_Stdio** host wraps the same `TsgcAI_MCP_Server` engine, reads newline-delimited JSON-RPC messages from standard input and writes responses to standard output (UTF-8). This is the transport used by local AI agents such as **Claude Code**, which spawn the server as a child process. No HTTP server, port or TLS configuration is required.
Register tools, prompts and resources on **TsgcAI_MCP_Server_Stdio.MCPServer** exactly as on the HTTP component, then call **Run** to start processing messages. Standard output is the protocol channel, so send any logging to standard error instead.

Reference

- Properties — 5 properties available on the component.
- Methods — 9 public methods.
- Events — 12 events you can handle.

MCP Server | Sessions

After a successful initialization between the server and the client, a new session is created. The session is always sent in the request and response between the client and the server to identify which session is being used in a request/response flow. When using the HTTP transport, the lifetime of a session is a single HTTP request and response, meaning that once the response has been sent, the session is closed.

Note: sessions only apply to clients that use the protocol version **2025-11-25** or older, which start with an `initialize` request. Clients that use the protocol version **2026-07-28** do not send `initialize` and are served statelessly: every request carries its own protocol version, client information and capabilities in `_meta`, no `Mcp-Session-Id` header is created, and the events **OnMCPSessionNew** and **OnMCPSessionEnd** are not raised for them. The *aSession* parameter of the request events is still available for modern requests, as a transient object that only lives during the request; use **aSession.Request.Era** to know the era of the request. See Protocol Versions.

The session object is passed on every event and to know when a session has been created or deleted, use the following events:

- **OnMCPSessionNew:** every time a new session id is created this event is called.
- **OnMCPSessionEnd:** when a session has ended, this event is called.

OnMCPSessionNew

The event is called when a new session is created. You can access the Session parameter to get the ID of the session, when the session expires, and more.

OnMCPSessionEnd

The event is called when a session has finished.

```
void __fastcall TForm1::OnMCPSessionEnd(TObject *Sender, const TsgcAI_MCP_Session *aSession)
{
    printf("#session_end: %s\n", aSession->Id.c_str());
}
void __fastcall TForm1::OnMCPSessionNew(TObject *Sender, const TsgcAI_MCP_Session *aSession)
{
    printf("#session_new: " + aSession->Id);
}
```

MCP Server | Protocol Versions

The MCP server supports the protocol versions **2026-07-28**, **2025-11-25**, **2025-06-18** and **2025-03-26** at the same time. No configuration is needed, the server detects the protocol era of every request and answers in the same era.

Protocol eras

- **Legacy era (2025-11-25 and older):** the client sends `initialize`, receives an `Mcp-Session-Id` and uses that session on every following request. This flow works exactly as in previous releases, see Sessions.
- **Modern era (2026-07-28):** there is no `initialize` and no session. Every request is self contained and carries the protocol version, the client information and the client capabilities in `params._meta`. The server processes each request statelessly, so no `Mcp-Session-Id` header is created or echoed.

A request is treated as modern when its `_meta` contains `io.modelcontextprotocol/protocolVersion` with a modern (or unknown) version. An `initialize` request, or a request without that field, follows the legacy flow. Inside any event handler you can check the era of the current request using **aSession.Request.Era** (`aimcpeModern` or `aimcpeLegacy`) and **aSession.Request.ProtocolVersion**.

server/discover

A 2026-07-28 client calls `server/discover` to learn which versions the server supports, its capabilities, its extensions and optional instructions. The server answers automatically using the values of **MCPOptions.ServerInfo**, **MCPOptions.Instructions** and **MCPOptions.Extensions**, plus the capabilities built from the registered tools, prompts and resources. The discover result is cacheable.

Modern requests supported by the server: `server/discover`, `tools/list`, `tools/call`, `prompts/list`, `prompts/get`, `resources/list`, `resources/read`, `resources/templates/list`, `completion/complete`, `subscriptions/listen`, the `tasks/*` methods and the notifications `notifications/progress` and `notifications/cancelled`. Methods that only exist in the legacy era (`ping`, `logging/setLevel`, `resources/subscribe`, `resources/unsubscribe`, ...) return error -32601.

HTTP headers validation

When a modern request arrives over HTTP, the server validates that the headers match the JSON body before the request is dispatched:

- **MCP-Protocol-Version** must be equal to `_meta.io.modelcontextprotocol/protocolVersion`.
- **Mcp-Method** must be equal to the JSON-RPC `method`.
- **Mcp-Name** must be equal to `params.name` (or `params.uri` for resources, or the task id for `tasks/*`). Base64 encoded values are decoded before the comparison and control characters are rejected.
- **Mcp-Param-*** headers are checked for `tools/call` requests when the tool declares `x-mcp-header` properties (see below).

A modern GET or DELETE request to the MCP endpoint returns HTTP 405, because the 2026-07-28 transport only uses POST. The **Origin** validation of the endpoint applies to modern requests too.

Error codes

Code	HTTP status	Meaning
`-32020`	400	Header mismatch. An MCP header does not match the value in the request body.
`-32021`	400	Missing required client capability. The request needs a capability the client did not declare in `_meta` (for example elicitation or the tasks extension).

-32022	400	Unsupported protocol version. The error data contains the list of supported versions so the client can retry with one of them.
-32601	404	Method not found, including legacy only methods called in the modern era.
-32602	200	Invalid params. Raise it from your handler, for example when a resource is not found.

To return a JSON-RPC error from any request handler, raise a **TsgcAI_MCP_Exception**. The error -32021 is sent automatically when a feature needs a client capability that the request did not declare, for example a multi round-trip request asking for elicitation, or a tool that requires the tasks extension.

Your handlers can check the client capabilities too. **HasClientCapability(aSession, aCapability)** returns True when the client declared the capability (in `_meta` for a 2026-07-28 request, in `initialize` for a legacy session), and **CheckClientCapability(aSession, aCapability)** raises the error -32021 when it is missing.

Caching, instructions and extensions

Modern list results and the discover result can be cached by the client. Configure the cache hints with **MCPOptions.Cache**:

- **Scope:** `aimcpcsPrivate` (default) or `aimcpcsPublic`, sent as `cacheScope`.
- **TTLMs:** time to live in milliseconds, sent as `ttlMs`. By default 60000.

MCPOptions.Instructions is a free text sent in the `server/discover` result to tell the model how to use the server. **MCPOptions.Extensions** is a raw JSON object with the extensions advertised in `server/discover`; the tasks extension is added automatically when **MCPOptions.Tasks.Enabled** is True.

x-mcp-header

A 2026-07-28 tool can ask the client to mirror some arguments of `tools/call` as HTTP headers, so proxies and gateways can route the request without parsing the body. Set the **XMCPHeader** property of the input schema property; it is published in `tools/list` as `x-mcp-header` and the client sends the argument as the header `Mcp-Param-{XMCP-Header}`. When the tool uses **RawInputSchema**, add `"x-mcp-header"` directly to the property in the JSON schema. The server checks that the header value matches the argument and answers -32020 if not.

MCP Server | Subscriptions

In the MCP protocol version **2026-07-28** there are no sessions, so a client that wants to receive notifications opens a long lived `subscriptions/listen` request. The request lists which notifications the client is interested in: tools, prompts and resources list changes, and updates of specific resource URIs. The server acknowledges the subscription and keeps the stream open, sending every matching notification until the client cancels it or the server closes it.

The server handles `subscriptions/listen` automatically, you only have to send the notifications. Legacy clients (2025-11-25 and older) keep receiving notifications through their session as before.

Configuration

- **MCPOptions.Subscriptions.Enabled:** enables the `subscriptions/listen` method. By default True. When False, the server answers -32601.
- **MCPOptions.Subscriptions.KeepAliveInterval:** interval in milliseconds between keep-alive comments written on an idle stream. By default 15000; set 0 to disable.

The acknowledgement only contains the notification types the server can deliver. Resource subscriptions are honored only when the server has resources registered.

Sending notifications

Use the same methods as in the legacy era. Every method sends the notification to the legacy sessions and also to every open subscription that asked for it, tagged with its subscription id:

- **SendNotificationToolsListChanged**: sends `notifications/tools/list_changed`.
- **SendNotificationPromptsListChanged**: sends `notifications/prompts/list_changed`.
- **SendNotificationResourcesListChanged**: sends `notifications/resources/list_changed`.
- **SendNotificationResourcesUpdated(aUri)**: sends `notifications/resources/updated` to the subscriptions that include that URI.

Closing the subscriptions

Before stopping the server, call **CloseSubscriptions**. Every open `subscriptions/listen` stream receives its final response and is closed gracefully, so the clients know the stream ended on purpose and do not treat it as a network failure. The method waits up to 3 seconds for the streams to finish. The component destructor calls it too.

Over HTTP/1.x each subscription keeps its HTTP connection open while it is active. The number of open subscriptions is available in the read only property **ListenersCount**, and **NotifySubscriptions(aMethod, aUri)** sends a notification to every subscription that watches it; *aUri* is only used by `notifications/resources/updated`.

MCP Server | Tasks

The tasks extension (`io.modelcontextprotocol/tasks`) of the MCP protocol version **2026-07-28** lets a server run a long `tools/call` in the background. Instead of waiting for the result, the server answers immediately with a task handle (`resultType` **task**) and the client polls it with `tasks/get`, sends requested input with `tasks/update` or cancels it with `tasks/cancel`. The server handles the three `tasks/*` methods automatically.

Configuration

- **MCPOptions.Tasks.Enabled:** enables the extension and advertises it in `server/discover`. By default False.
- **MCPOptions.Tasks.TTL:** time to live of a task in milliseconds, after which it is purged. By default 3600000; 0 means unlimited.
- **MCPOptions.Tasks.PollInterval:** suggested polling interval in milliseconds sent to the client. By default 1000.
- **MCPOptions.Tasks.MaxTasks:** maximum number of live tasks. By default 1000.

A task is only created when the client declared the tasks extension in the `_meta` of that request. Use the **TaskSupport** property of the tool to control it: `aimcptsForbidden` never creates a task, `aimcptsOptional` creates it when the client supports it, and `aimcptsRequired` rejects clients without the extension with error -32021.

Creating a task

Inside **OnMCPRequestTool**, call **CreateTask(aSession, aResponse)**. If it returns a task object, the response is converted into a task handle: start the work in a background thread and exit the handler. If it returns nil (tasks disabled, legacy client, extension not declared, tool forbids tasks or MaxTasks reached), run the tool synchronously as usual.

TsgcAI_MCP_Task

The task object is owned by the server and is thread safe, do not free it. Its methods return False when the task is already in a terminal state (completed, failed or cancelled) or has expired.

- **Complete(aToolsCallResponse) / Complete(aResultJSON):** finishes the task with a `tools/call` result, passing a response object or the raw JSON of the result.
- **Fail(aCode, aMessage, aDataJSON):** finishes the task with a JSON-RPC error.
- **SetStatusMessage(aMessage):** updates the status message the client reads with `tasks/get`.
- **RequireInput(aInputRequests):** moves the task to **input_required** with a list of inputs built with **TsgcAI_MCP_InputRequests** (see Multi Round-Trip Requests). The object is copied, so free it afterwards. Keys must be unique during the lifetime of the task.
- Read only properties: **TaskId**, **Method**, **Name**, **Principal**, **Status**, **StatusMessage** and **IsCancelled**.

Task events

- **OnMCPTaskCancel(Sender, aTask):** the client sent `tasks/cancel`. Stop the background work; **IsCancelled** is True from now on.
- **OnMCPTaskUpdate(Sender, aTask, aInputResponses):** the client sent `tasks/update` with the answers to the inputs requested by **RequireInput**. *aInputResponses* is a raw JSON object with one member per key.

Both events are published, assign them in the Object Inspector or in code.

The read only property **TasksCount** returns the number of tasks stored by the server, working or finished; expired tasks are purged before counting.

Tasks are bound to the principal returned by **DoGetRequestPrincipal**, so a client can only read, update or cancel its own tasks. An unknown task returns error -32602 "Task not found" and an expired one "Failed to retrieve task: Task has expired".

MCP Server | Logging and Progress

While a request is being processed, the server can send log messages (`notifications/message`) and progress notifications (`notifications/progress`) to the client that made the request. Use the overloads of **SendLogMessage** and **SendNotificationProgress** that receive the session of the current request.

- **SendLogMessage(aSession, aLevel, aLogger, aData):** sends a log entry related to the current request.
- **SendNotificationProgress(aSession, aProgressToken, aProgress, aTotal, aMessage):** reports the progress of the current request. The progress token is sent by the client in the request `_meta` and is available in **aRequest.ProgressToken**.

2026-07-28 clients

Modern requests have no session, so the notifications are streamed in the response of the same request. When the client accepts `text/event-stream`, the HTTP response is sent as a Server-Sent Events stream with every notification followed by the final result. Over stdio the notifications are written before the response.

- A log message is only sent when the client requested logs for that request with `logLevel` in `_meta`, and the level of the message is equal to or higher than the requested level.
- A progress notification is only sent when the request contains a progress token.

Legacy clients

For 2025-11-25 and older clients the overloads behave like the methods without session: the log level is the one set by the client with `logging/setLevel`, and the notifications are delivered through the session streams, which requires **TransportOptions.HttpStreamable.Enabled**.

MCP Server | Tools

The **TsgcWSAPIServer_MCP** is the high-level component that exposes Model Context Protocol (MCP) server features over sgcWebSockets. When **MCP clients request tools** according to the MCP Server Tools specification, this component orchestrates the HTTP gateway, session lifecycle and JSON-RPC serialization so that your Delphi code can focus on implementing business logic.

Incoming HTTP requests that hit the MCP endpoint defined in **EndpointOptions.Endpoint** are intercepted by **TsgcWSServer_API_MCP**. The component creates (or resumes) an MCP session, forwards the request to the underlying TsgcAI_MCP_Server engine and finally writes the JSON-RPC response back to the HTTP client while preserving the mcp-session-id header required by the specification.

Tools List

The server keeps an in-memory catalogue of tools in **TsgcAI_MCP_ToolsList**. Tools are guaranteed to be unique by name, expose descriptions and provide a JSON-Schema-like input description that is emitted in the response to the specification's tools.list method. When a client invokes tools.list, TsgcWSAPIServer_MCP loads the request, serializes the current catalogue and sends it back with a 200 HTTP status code.

Example code that publishes a simple calculator tool:

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    TsgcAI_MCP_Tool* oTool;
    // Add a tool named "math.add"
    oTool = MCPServer->Tools->AddTool("math.add", "Adds two numbers");
    // Define input schema properties
    oTool->InputSchema->Properties->AddProperty("a", true, aimcpjtNumber, "Left operand");
    oTool->InputSchema->Properties->AddProperty("b", true, aimcpjtNumber, "Right operand");
}
```

A compliant tools.list response sent to the client will mirror the MCP JSON schema payload. For example:

```
{
  "jsonrpc": "2.0",
  "id": "42",
  "result": {
    "tools": [
      {
        "name": "math.add",
        "description": "Adds two numbers",
        "inputSchema": {
          "type": "object",
          "required": ["a", "b"],
          "properties": {
            "a": { "type": "number", "description": "Left operand" },
            "b": { "type": "number", "description": "Right operand" }
          }
        }
      }
    ]
  }
}
```

Array items, enum values and the integer type

The structured property builder also supports arrays, enumerations and the JSON Schema **integer** type. Create the property, set its **Items** (the array element schema) or **Enum** (the list of accepted values), then add it to the schema:

The **Enum** list only accepts string values. If a client needs an enum that also allows `null`, or a non-string enum, describe it with **RawInputSchema** instead (see below).

Complete JSON Schema (RawInputSchema)

The property builder covers the common cases, but the Model Context Protocol allows a tool's **inputSchema** to be any valid JSON Schema. When a client needs something the builder does not expose — a nested object, **additionalProperties**, a nullable type expressed as `["string", "null"]`, or an enum that includes `null` — assign a complete schema directly to **RawInputSchema**. When it is assigned, it is sent verbatim as the tool's `inputSchema` and the structured **InputSchema** builder is ignored for that tool.

Tool Request

When a client issues a tools.call JSON-RPC request, **TsgcWSAPIServer_MCP** hydrates the strongly-typed request object (including tool name and the provided arguments) before raising the **OnMCPRequestTool** event. Your handler populates the response payload which is then serialized back to the client, alongside a success HTTP status code.

A typical handler looks like this:

```
void __fastcall TMainForm::MCPServerMCPRequestTool(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession,
    const TsgcAI_MCP_Request_ToolsCall *ARequest,
    const TsgcAI_MCP_Response_ToolsCall *AResponse)
{
    double LA, LB;
    if (ARequest->Params->Name == "math.add")
    {
        LA = ARequest->Params->Arguments->Node["a"]->AsNumber;
        LB = ARequest->Params->Arguments->Node["b"]->AsNumber;
        AResponse->Result->Content->AddText(
            Format("Sum = %.2f", ARRAYOFCONST((LA + LB)))
        );
    }
    else
    {
        AResponse->Result->IsError = true;
    }
}
```

The generated JSON-RPC response will follow the spec's tools.call schema, wrapping one or more content parts and optionally a structured payload. Because the component clears the response and sets the HTTP code for you, no additional plumbing is required.

Tool annotations, icons and task execution hints

Beyond name, description and input schema, a tool can advertise hints that help a client decide how to present or invoke it. **Annotations.ReadOnlyHint** tells the client the tool never modifies state, and **Annotations.Title** gives the annotations block its own human-readable title (distinct from the tool's Title). **Icons** is a raw JSON array of MCP icon descriptors displayed by the client next to the tool. **TaskSupport** declares whether the tool can run as a long-lived MCP task (*forbidden*, *optional* or *required*), emitted as `execution.taskSupport` in the tools.list response.

Important: per the MCP icon schema, **sizes** must be a JSON array of strings (for example `["48x48"]`), even for a single size. A bare string is rejected by MCP client SDKs.

Availability: RawInputSchema, Items, Enum, the integer type, ReadOnlyHint, annotation Title, Icons and TaskSupport are available from Delphi and C++Builder (same class and property names, via the generated header). Support in the .NET edition is planned for a future release.

Public API surface

Several helper classes surface the tool catalogue and the event response helpers. The following lists summarise the methods that are typically called from production code.

TsgcAI_MCP_ToolsList

- **AddTool(const Name, Description):** creates or replaces a tool entry and returns the underlying **TsgcAI_MCP_Tool** so you can configure schemas and defaults.
- **Clear:** removes all registered tools. Call this before rebuilding the catalogue when configuration is loaded from disk.
- **Count / Items[Index]:** expose the list contents for enumeration, diagnostics or custom serialisation.

TsgcAI_MCP_Tool

- **InputSchema.Properties.AddProperty(Name, Required, JsonType, Description):** documents the JSON input payload that will be advertised through **tools.list**.
- **InputSchema.Properties.AddProperty(Property):** overload that accepts a fully-built **TsgcAI_MCP_Tool_InputSchema_Property**, so you can set **Items** (array element schema) or **Enum** (accepted values) before adding it. The JSON type list also includes **aimcpjtInteger** for the JSON Schema integer type.
- **InputSchema.Properties.Clear:** rebuild the schema on the fly when your application refreshes tool capabilities at runtime.
- **Description:** writable property that stores the human-readable summary shared with clients.
- **RawInputSchema:** assign a complete `IsgcJSON` JSON Schema to be sent verbatim as **inputSchema**, bypassing the structured builder. Use it for nested objects, additionalProperties, nullable types or an enum that includes null.
- **Annotations.ReadOnlyHint / Annotations.Title:** flags the tool as read-only and gives its annotations block a title, alongside the existing DestructiveHint, IdempotentHint and OpenWorldHint.
- **Icons:** a raw `IsgcJSON` array of MCP icon descriptors shown by the client next to the tool.
- **TaskSupport:** declares *forbidden*, *optional* or *required* support for MCP task-based execution, emitted as **execution.taskSupport**.

TsgcAI_MCP_Response_ToolsCall

- **Result.Content.AddText(Value):** appends text blocks to the tool response. Multiple calls produce multi-part responses.
- **Result.Content.AddImage(Data, MimeType):** appends image blocks to the tool response. Multiple calls produce multi-part responses
- **Result.Content.AddAudio(Data, MimeType):** appends audio blocks to the tool response. Multiple calls produce multi-part responses.
- **Result.Content.AddEmbeddedResource(Uri, Title, Text, MimeType):** appends embedded resource blocks to the tool response. Multiple calls produce multi-part responses.
- **Result.Content.AddResourceLink(Uri, Name, MimeType):** appends resource link blocks to the tool response. Multiple calls produce multi-part responses.
- **Result.Content.Clear:** resets any previously generated output before you append new content.
- **Result.IsError:** set this flag to **True** when the tool call cannot be fulfilled so the framework serialises an MCP error object.

Advanced example

The next snippet builds a geo lookup tool that validates inputs, returns multiple text blocks and flips **IsError** when the parameters are invalid.

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    TsgcAI_MCP_Tool *tool = MCPServer->Tools->AddTool("geo.locate", "Returns coordinates for a city and country");
    tool->InputSchema->Properties->Clear();
    tool->InputSchema->Properties->AddProperty("city", true, aimcpjtString, "City to search");
    tool->InputSchema->Properties->AddProperty("country", false, aimcpjtString, "Optional ISO country code");
}
void __fastcall TMainForm::MCPServerMCPRequestTool(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession,
    const TsgcAI_MCP_Request_ToolsCall *ARequest,
    const TsgcAI_MCP_Response_ToolsCall *AResponse)
{
    if (!SameText(ARequest->Params->Name, "geo.locate"))
        return;
    UnicodeString city = ARequest->Params->Arguments->Node["city"]->AsString();
    UnicodeString country = ARequest->Params->Arguments->Node["country"]->AsString();
    AResponse->Result->Content->Clear();
    if (city.IsEmpty())
    {
        AResponse->Result->Content->AddText("City cannot be empty.");
        AResponse->Result->IsError = true;
        return;
    }
    AResponse->Result->Content->AddText("City: " + city);
    if (!country.IsEmpty())
        AResponse->Result->Content->AddText("Country: " + country);
    AResponse->Result->Content->AddText("Coordinates: " + LookupCityCoordinates(city, country));
}
```

MCP Server | Prompts

In MCP, *prompts* are reusable templates or workflows the server exposes. They are user-controlled (clients choose which prompt to invoke). The two main methods are:

- **prompts/list:** Requests the list of all available prompt definitions exposed by the server. It's handled internally by the TsgcWSAPIServer_MCP component.
- **prompts/get:** Retrieves a specific prompt's content and optional metadata, typically a message sequence describing how the client should interact. The event **OnMCPRequestPrompt** is called when a new request is received by the server.

Additionally, if the server sets prompts.listChanged = true in its declared capabilities, it may send notifications like notifications/prompts/list_changed when the prompt catalog updates.

When a client requests prompts/get, it sends arguments (if any) and expects a sequence of messages (user or assistant roles) with content blocks (usually text, but could embed resources or images).

If something is wrong (missing prompt, invalid args), the server will return JSON-RPC error codes such as -32602 (Invalid params) or -32603 (Internal error).

Prompts List

The server keeps an in-memory catalogue of prompts in **TsgcAI_MCP_PromptsList**. Prompts are guaranteed to be unique by name, expose descriptions and provide a JSON-Schema-like input description that is emitted in the response to the specification's prompts.list method. When a client invokes prompts.list, TsgcWSAPIServer_MCP loads the request, serializes the current catalogue and sends it back with a 200 HTTP status code.

Example code that publishes a code review prompt:

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    TsgcAI_MCP_Prompt* oPrompt;
    // Clear all existing prompts
    MCPServer->Prompts->Clear();
    // Create a new prompt named "CodeReview"
    oPrompt = MCPServer->Prompts->AddPrompt(
        "CodeReview",
        "Asks the LLM to analyze code quality and suggest improvements"
    );
    // Add argument 'code'
    oPrompt->Arguments->AddArgument(
        "code",
        "The code to review",
        true  // required
    );
}
```

Prompt Request

When a client issues a prompts.get JSON-RPC request, **TsgcWSAPIServer_MCP** hydrates the strongly-typed request object (including the prompt name and the provided arguments) before raising the **OnMCPRequestPrompt** event. Your handler populates the response payload which is then serialized back to the client, alongside a success HTTP status code.

A typical handler looks like this:

```
void __fastcall TFRMMCPServer::MCPServerMCPRequestPrompt(
    TObject *Sender,
```

```
    const TsgcAI_MCP_Session *aSession,
    const TsgcAI_MCP_Request_PromptsGet *aRequest,
    const TsgcAI_MCP_Response_PromptsGet *aResponse)
{
    if (aRequest->Params->Name == "CodeReview")
    {
        aResponse->Result->Description = "Code review prompt";
        aResponse->Result->Messages->AddText(
            "user",
            "Please review this Delphi code: ShowMessage('Hello World');"
        );
    }
}
```

Public API surface

Prompts are modelled through a handful of helper classes. The following methods are used most frequently when curating prompt templates and when composing a **prompts/get** response.

TsgcAI_MCP_PromptsList

- **AddPrompt(const Name, Description):** registers a new prompt entry and returns a **TsgcAI_MCP_Prompt** object so that arguments and messages can be configured.
- **Clear:** wipes the catalogue, handy during start-up when prompts are loaded from configuration files.
- **Count / Items[Index]:** provide access to the list contents for diagnostics, UI visualisation or manual serialisation.

TsgcAI_MCP_Prompt

- **Arguments.AddArgument(Name, Description, Required):** declares the structured inputs the prompt accepts.
- **Arguments.Clear:** rebuilds the argument list when your prompt changes at runtime.
- **Messages.AddText(Role, Text):** pre-seeds conversational turns that are sent back to the client as part of the prompt definition.

TsgcAI_MCP_Response_PromptsGet

- **Result.Description:** short human-readable summary that accompanies the prompt payload.
- **Result.Messages.AddText(Role, Text):** injects additional messages dynamically when the prompt is fetched.
- **Result.Messages.AddImage(Data, MimeType):** appends image blocks to the prompt response. Multiple calls produce multi-part responses
- **Result.Messages.AddAudio(Data, MimeType):** appends audio blocks to the prompt response. Multiple calls produce multi-part responses.
- **Result.Messages.AddEmbeddedResource(Uri, Title, Text, MimeType):** appends embedded resource blocks to the prompt response. Multiple calls produce multi-part responses.
- **Result.Messages.Clear:** reset the response before reusing the object.

Advanced example

The next example publishes a prompt that guides an LLM through a triage workflow. It highlights optional arguments and how to return multi-step instructions in the handler.

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
```

```
    TsgcAI_MCP_Prompt *prompt;
    MCPServer->Prompts->Clear();
    prompt = MCPServer->Prompts->AddPrompt("IssueTriage", "Collect details before escalating support tickets");
    prompt->Arguments->Clear();
    prompt->Arguments->AddArgument("summary", "One-line description supplied by the user", true);
    prompt->Arguments->AddArgument("customerPriority", "Optional priority label", false);
    prompt->Messages->AddText("system", "You are a support assistant that triages technical issues.");
}
void __fastcall TMainForm::MCPServerMCPRequestPrompt(
    TObject *Sender,
    const TsgcAI_MCP_Session *aSession,
    const TsgcAI_MCP_Request_PromptsGet *aRequest,
    const TsgcAI_MCP_Response_PromptsGet *aResponse)
{
    if (!SameText(aRequest->Params->Name, "IssueTriage"))
        return;
    aResponse->Result->Description = "Guided checklist for support analysts.";
    aResponse->Result->Messages->Clear();
    aResponse->Result->Messages->AddText("user", "Review the ticket summary and ask clarifying questions.");
    aResponse->Result->Messages->AddText("assistant", "Acknowledge the request and confirm the escalation path.")
    if (aRequest->Params->Arguments->Node["customerPriority"]->AsString() != "")
        aResponse->Result->Messages->AddText("assistant", "Adjust SLA checks based on the provided customer prior
}
```

MCP Server | Resources

In MCP, **resources** represent addressable data objects that the server exposes such as files, database records, generated documents, or dynamic API outputs.

They are *client-controlled*, meaning the client can decide which resource to request and how to interpret the content.

The two main methods involved are:

- **resources/list:** Requests the list of all available resource definitions the server exposes. It's handled internally by the TsgcWSAPIServer_MCP component.
- **resources/read:** Retrieves the content and optional metadata of a specific resource. The event **OnMCPRequestResource** is called when a new request is received by the server. A resource typically includes:
  - **uri:** resource identifier.
  - **mimeType:** content type (text/plain, application/json, image/png, etc.).
  - **data or content:** the actual payload (text or binary).

Additionally, if the server sets resources.listChanged = true in its declared capabilities, it may send notifications like notifications/resources/list_changed when the resource catalog updates.

Resources List

The server keeps an in-memory catalogue of resources in **TsgcAI_MCP_ResourcesList**. Resources are guaranteed to be unique by uri, expose descriptions and provide a JSON-Schema-like input URI that is emitted in the response to the specification's resources.list method. When a client invokes resources.list, **TsgcWSAPIServer_MCP** loads the request, serializes the current catalogue and sends it back with a 200 HTTP status code.

Example code that publishes a file resource:

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    // Clear any previously registered resources
    MCPServer->Resources->Clear();
    // Register a Rust source file resource
    MCPServer->Resources->AddResource(
        "file:///project/src/main.rs",        // URI
        "main.rs",                            // Name
        "Rust Software Application Main File",// Title
        "Primary application entry point",    // Description
        "text/x-rust"                         // MIME type
    );
}
```

Resources Request

When a client issues a resource.read JSON-RPC request, **TsgcWSAPIServer_MCP** hydrates the strongly-typed request object (including uri resource, name and the provided arguments) before raising the **OnMCPRequestResource** event. Your handler populates the response payload which is then serialized back to the client, alongside a success HTTP status code.

A typical handler looks like this:

```
void __fastcall TMainForm::OnMCPRequestResource(
    TObject *Sender,
    const TsgcAI_MCP_Session *aSession,
    const TsgcAI_MCP_Request_ResourcesRead *aRequest,
```

```
    const TsgcAI_MCP_Response_ResourcesRead *aResponse)
{
    if (aRequest->Params->Uri == "file:///project/src/main.rs")
    {
        aResponse->Result->Contents->AddContentText(
            "file:///project/src/main.rs",         // URI
            "main.rs",                             // Name
            "Rust Software Application Main File", // Description
            "text/x-rust",                         // MIME type
            "fn main() {\n    println!(\"Hello world!\");\n}" // Content
        );
    }
}
```

Public API surface

Resources combine a catalogue entry (**TsgcAI_MCP_Resource**) and the streaming helpers used inside **OnMCPRequestResource**. These are the methods you will reach for most often.

TsgcAI_MCP_ResourcesList

- **AddResource(Uri, Name, Title, Description, MimeType):** registers a static or dynamically generated resource.
- **Clear:** removes every resource definition, useful before rebuilding the list from configuration files.
- **Count / Items[Index]:** expose the catalogue for enumeration or diagnostic views.

TsgcAI_MCP_Resource

- **MimeType / Title / Description:** writable properties that describe the resource payload returned to the client.
- **Name / Uri:** use these properties to update identifiers without re-registering the resource entry.

TsgcAI_MCP_Response_ResourcesRead

- **Result.Contents.AddContentText(Uri, Name, Title, MimeType, Text):** sends a text payload back to the caller.
- **Result.Contents.AddContentBlob(Uri, Name, Title, MimeType, Text):** sends a text payload back to the caller.
- **Result.Contents.Clear:** resets the outgoing payload before reusing the response object.
- **Result.IsError:** flag the response as an error when the resource cannot be produced.

Advanced example

The example below publishes a JSON resource that returns aggregated metrics. It demonstrates how to validate the incoming URI and craft different responses depending on the request.

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    MCPServer->Resources->Clear();
    MCPServer->Resources->AddResource(
        "metrics://daily",
        "DailyMetrics",
        "Daily KPI snapshot",
        "Summaries generated each night",
        "application/json");
}
void __fastcall TMainForm::MCPServerMCPRequestResource(
    TObject *Sender,
    const TsgcAI_MCP_Session *aSession,
```

```
    const TsgcAI_MCP_Request_ResourcesRead *aRequest,
    const TsgcAI_MCP_Response_ResourcesRead *aResponse)
{
    UnicodeString targetUri = aRequest->Params->Uri.LowerCase();
    aResponse->Result->Contents->Clear();
    if (targetUri == "metrics://daily")
    {
        aResponse->Result->Contents->AddContentText(
            "metrics://daily",
            "DailyMetrics",
            "Daily KPI snapshot",
            "application/json",
            BuildMetricsPayload(Date()));
        return;
    }
    aResponse->Result->IsError = true;
    aResponse->Result->Contents->AddContentText(
        aRequest->Params->Uri,
        "UnknownResource",
        "Not found",
        "text/plain",
        "Resource is not published by this server.");
}
```

MCP Server | Roots

Roots mark the filesystem locations a client allows the server to explore. Each entry is a file:// URI with an optional display name, letting MCP automations honour project boundaries while the user keeps control of what is shared. Servers query the catalogue by issuing roots/list and, when the client announces listChanged, receive notifications/roots/list_changed so they can refresh cached views.
Because roots are client-owned, the server always acts as the requester. The roots capability must be present in the client's initialise payload before you attempt a discovery call; otherwise RequestRootsList raises CapabilityNotSupported.

Requesting the root catalogue

TsgcWSAPIServer_MCP exposes a helper that serialises roots/list and routes the reply back to your code:

- **Request helper:** call RequestRootsList(Session) once the MCP session advertises the roots capability. The method assigns an identifier automatically (unless you provide one) and persists the pending request so the response can be correlated.
- **Response event:** handle OnMCPResponseRootsList(Sender, Session, Request, Response). The event fires when the client answers with the available directories. Inspect Response.Roots or log Response.Write to examine the JSON payload.
- **Change detection:** if the client later sends notifications/roots/list_changed, submit another RequestRootsList so your server side cache stays in sync.

Client replies and protocol eras

The reply of the client is matched with the pending request and raises **OnMCPResponseRootsList**. Over HTTP the client posts the reply to the MCP endpoint and the server answers 204 without body. A client that has no handler for the request answers with the JSON-RPC error -32601 (Method not found). An error reply also resolves the pending request: the server answers 204 and raises the same **OnMCPResponseRootsList** event with **aResponse.IsError** set to True. Read **ErrorCode**, **ErrorMessage** and **ErrorData** of the response to know why the request failed, and check **IsError** before reading the result.
This request only exists in the legacy era (2025-11-25 and older). Clients of the protocol version 2026-07-28 have no session, so ask them for the same input from the tool, prompt or resource handler with aResponse.InputRequired.AddRoots. See Multi Round-Trip Requests.

Sample code

The snippets below request roots as soon as a session is established and log the returned entries.

```
void __fastcall TFRMMCPServer::FormCreate(TObject *Sender)
{
    MCPServer->OnMCPSessionNew = MCPServerSessionNew;
    MCPServer->OnMCPResponseRootsList = MCPServerRootsList;
}

void __fastcall TFRMMCPServer::MCPServerSessionNew(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession)
{
    if (ASession->ClientCapabilities->Roots->Enabled)
    {
        System::UnicodeString requestId = MCPServer->RequestRootsList(ASession);
        MemoLog->Lines->Add("roots/list sent id " + requestId);
    }
}

void __fastcall TFRMMCPServer::MCPServerRootsList(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession,
    const TsgcAI_MCP_Request_RootsList *ARequest,
    const TsgcAI_MCP_Response_RootsList *AResponse)
{
    MemoLog->Lines->Add(
        Format("roots/list returned %d entries", ARRAYOFCONST((AResponse->Roots->Count)))
    );
```

```
    MemoLog->Lines->Add("roots/list payload " + AResponse->Write());
}
```

MCP Server | Sampling

Sampling lets the server ask the client to run an LLM generation on its behalf. Requests describe a conversation history, optional system prompt, model preferences and token budgets. The client keeps control of which provider is used, performs any human-in-the-loop reviews and finally returns the approved completion.
Clients must declare the sampling capability during initialize; otherwise a sampling attempt is rejected with a capability error. The specification encourages user approval and allows text, image or audio content blocks inside the conversational history.

Issuing a sampling request

TsgcWSAPIServer_MCP streamlines the sampling/createMessage flow with a helper and a response event:

- **Request helper:** build a TsgcAI_MCP_Request_SamplingCreateMessage, populate Params.Messages, Params.ModelPreferences, Params.SystemPrompt and optionally Params.MaxTokens. Pass it to RequestSamplingCreateMessage(Session, Request); the method stamps an id (if needed) and records the pending call.
- **Response event:** handle OnMCPResponseSamplingCreateMessage(Sender, Session, Request, Response). Inspect Response.Role, Response.Content, Response.Model and Response.StopReason to continue your workflow.
- **Error handling:** declined or cancelled requests arrive as JSON-RPC errors, so wrap the helper in try/except and monitor OnMCPException for logging.

Client replies and protocol eras

The reply of the client is matched with the pending request and raises **OnMCPResponseSamplingCreateMessage**. Over HTTP the client posts the reply to the MCP endpoint and the server answers 204 without body. A client that has no handler for the request answers with the JSON-RPC error -32601 (Method not found). An error reply also resolves the pending request: the server answers 204 and raises the same **OnMCPResponseSamplingCreateMessage** event with **aResponse.IsError** set to True. Read **ErrorCode**, **ErrorMessage** and **ErrorData** of the response to know why the request failed, and check **IsError** before reading the result.
This request only exists in the legacy era (2025-11-25 and older). Clients of the protocol version 2026-07-28 have no session, so ask them for the same input from the tool, prompt or resource handler with aResponse.InputRequired.AddSampling. See Multi Round-Trip Requests.

Sample code

The snippets below request a short summary from the client and print the assistant reply.

```
void __fastcall TFRMMCPServer::FormCreate(TObject *Sender)
{
    MCPServer->OnMCPResponseSamplingCreateMessage = MCPServerSamplingResponse;
}
void __fastcall TFRMMCPServer::RequestTicketSummary(
    const TsgcAI_MCP_Session *ASession,
    const System::UnicodeString ATicketText)
{
    if (!ASession->ClientCapabilities->Sampling->Enabled)
        return;
    std::unique_ptr TsgcAI_MCP_Request_SamplingCreateMessage> request(
        new TsgcAI_MCP_Request_SamplingCreateMessage());
    request->Params->Messages->AddTextMessage(
        "user",
        "Summarise the following ticket: " + ATicketText
    );
    request->Params->SystemPrompt = "You are a concise support assistant.";
    request->Params->ModelPreferences->Hints->AddHint("claude-3-sonnet");
    request->Params->ModelPreferences->SpeedPriority = 0.6;
    request->Params->ModelPreferences->IntelligencePriority = 0.8;
    request->Params->HasMaxTokens = true;
    request->Params->MaxTokens = 200;
    MemoLog->Lines->Add(
        "sampling/createMessage id " +
        MCPServer->RequestSamplingCreateMessage(ASession, request.get())
    );
```

```
}
void __fastcall TFRMMCPServer::MCPServerSamplingResponse(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession,
    const TsgcAI_MCP_Request_SamplingCreateMessage *ARequest,
    const TsgcAI_MCP_Response_SamplingCreateMessage *AResponse)
{
    if (dynamic_cast <TsgcAI_MCP_Response_Result_Content_Text*>(AResponse->Content) != nullptr)
    {
        auto *text = static_cast <TsgcAI_MCP_Response_Result_Content_Text*>(AResponse->Content);
        MemoLog->Lines->Add("assistant: " + text->Text);
    }
    MemoLog->Lines->Add(
        "model: " + AResponse->Model + " reason: " + AResponse->StopReason
    );
}
```

MCP Server | Elicitation

Elicitation lets the server ask the client to collect structured input from the user mid-workflow. The request contains a human-friendly message plus a constrained JSON schema describing the fields that must be captured. The client renders an appropriate form, validates the answer locally, and returns whether the user accepted, declined, or cancelled the prompt.
The client must advertise the elicitation capability during initialize. Schemas are intentionally limited to flat objects with primitive properties (string, number, boolean or enum) to make rendering straightforward and to keep sensitive data out of scope.

Requesting user input

TsgcWSAPIServer_MCP provides a helper for elicitation/create and raises a response event when the user finishes interacting:

- **Request helper:** create TsgcAI_MCP_Request_ElicitationCreate, set Params._Message, and describe the expected payload through Params.RequestedSchema. Call RequestElicitationCreate(Session, Request) to send it to the client.
- **Response event:** handle OnMCPResponseElicitationCreate(Sender, Session, Request, Response). Inspect Response.Action (accept, decline or cancel) and parse Response.Content when the user accepted.
- **Validation:** rejected requests arrive as JSON-RPC errors. Always check Action before consuming any content and be prepared to ask again if the user cancelled.

Client replies and protocol eras

The reply of the client is matched with the pending request and raises **OnMCPResponseElicitationCreate**. Over HTTP the client posts the reply to the MCP endpoint and the server answers 204 without body. A client that has no handler for the request answers with the JSON-RPC error -32601 (Method not found). An error reply also resolves the pending request: the server answers 204 and raises the same **OnMCPResponseElicitationCreate** event with **aResponse.IsError** set to True. Read **ErrorCode**, **ErrorMessage** and **ErrorData** of the response to know why the request failed, and check **IsError** before reading the result.
This request only exists in the legacy era (2025-11-25 and older). Clients of the protocol version 2026-07-28 have no session, so ask them for the same input from the tool, prompt or resource handler with aResponse.InputRequired.AddElicitation. See Multi Round-Trip Requests.

Sample code

The snippets below request contact information, wait for the reply, and log the resulting action plus payload.

```
void __fastcall TFRMMCPServer::FormCreate(TObject *Sender)
{
    MCPServer->OnMCPResponseElicitationCreate = MCPServerElicitationResponse;
}
void __fastcall TFRMMCPServer::RequestContactDetails(
    const TsgcAI_MCP_Session *ASession)
{
    if (!ASession->ClientCapabilities->Elicitation->Enabled)
        return;
    std::unique_ptr <TsgcAI_MCP_Request_ElicitationCreate> request(
        new TsgcAI_MCP_Request_ElicitationCreate());
    request->Params->_Message = "Please confirm your contact information.";
    request->Params->RequestedSchema->Title = "Contact information";
    request->Params->RequestedSchema->Description =
        "Used to keep you updated about this ticket.";
    auto *name = request->Params->RequestedSchema->Properties->AddProperty("name");
    name->_Type = "string";
    name->Title = "Full name";
    name->Required = true;
    auto *email = request->Params->RequestedSchema->Properties->AddProperty("email");
    email->_Type = "string";
    email->Title = "Email address";
    email->Format = "email";
    email->Required = true;
    auto *updates = request->Params->RequestedSchema->Properties->AddProperty("updates");
    updates->_Type = "boolean";
```

```
    updates->Title = "Receive status updates";
    updates->Description = "Tick to receive notifications when the ticket changes.";
    updates->DefaultBoolean = false;
    MemoLog->Lines->Add(
        "elicitation/create id " +
        MCPServer->RequestElicitationCreate(ASession, request.get())
    );
}
void __fastcall TFRMMCPServer::MCPServerElicitationResponse(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession,
    const TsgcAI_MCP_Request_ElicitationCreate *ARequest,
    const TsgcAI_MCP_Response_ElicitationCreate *AResponse)
{
    MemoLog->Lines->Add("elicitation action: " + AResponse->Action);
    if (AResponse->Action.LowerCase() == "accept" && AResponse->Content != nullptr)
        MemoLog->Lines->Add("payload: " + AResponse->Content->Text);
}
```

MCP Server | Multi Round-Trip Requests

In the MCP protocol version **2026-07-28** the server cannot send requests to the client (elicitation, sampling or roots) in the middle of a tool call, because there is no session. Instead, the server answers the request with `resultType` **input_required**, listing the inputs it needs. The client collects them and repeats the same request with the answers in `inputResponses`. This is called a Multi Round-Trip Request (MRTR) and is supported by `tools/call`, `prompts/get` and `resources/read`.

Asking for input

Inside **OnMCPRequestTool**, **OnMCPRequestPrompt** or **OnMCPRequestResource**, add the inputs you need to **aResponse.InputRequired** and exit the handler. Each input has a key, which is used later to read the answer.

- **AddElicitation(aKey, aMessage, aRequestedSchemaJSON):** asks the user to fill a form described by a JSON schema.
- **AddElicitationURL(aKey, aMessage, aURL):** asks the user to open a URL (URL mode elicitation).
- **AddSampling(aKey, aMessagesJSON, aSystemPrompt, aMaxTokens):** asks the client LLM for a completion.
- **AddRoots(aKey):** asks the client for its list of roots.
- **AddRequest(aKey, aMethod, aParamsJSON):** adds any other client request with raw JSON params.

Use **aResponse.InputRequired.RequestState** to store your own application state (for example the step of a wizard). The server signs it with HMAC-SHA256 and sends it to the client as `requestState`. When the client repeats the request, the state is verified (signature, expiration, method, name and principal) before your handler is called, and it is available in **aRequest.Params.RequestState**. A state that fails the verification is rejected with -32602 "Invalid requestState".

Reading the answers

When the client repeats the request, **aRequest.HasInputResponse(aKey)** returns True for every answered input and **aRequest.InputResponse(aKey)** returns the answer as raw JSON text (an empty string if there is no answer for that key).

Capabilities

Before sending **input_required**, the server checks that the client declared the needed capabilities in the request `_meta`: elicitation (and elicitation url for URL mode), sampling or roots. If a capability is missing, the client receives error -32021.

MRTR options

- **MCPOptions.MRTR.Secret:** key used to sign the request state. If empty, a random secret is created for each server instance, so the states are only valid for that instance. Set the same secret on every node when the server runs behind a load balancer.
- **MCPOptions.MRTR.StateTTL:** validity of the request state in seconds. By default 600.

The request state is bound to a principal returned by the protected virtual method **DoGetRequestPrincipal(aSession)** of the MCP server engine (**TsgcAI_MCP_Server**). By default it returns an empty string; override it to return the authenticated user, so a state created for one user cannot be reused by another.

Legacy clients

Multi round-trip requests only work with 2026-07-28 clients. If a handler adds inputs while serving a legacy client, the server answers with error -32603 "input_required needs protocol 2026-07-28". To support both eras, check **aSession.Request.Era** and use the classic server requests (Elicitation, Sampling, Roots) for legacy sessions.

TsgcWSAPIClient_MCP

TsgcWSAPIClient_MCP component.

Introduction

TsgcWSAPIClient_MCP implements a Model Context Protocol (MCP) consumer on top of sgcWebSockets networking components. It takes care of session negotiation, JSON-RPC request/response marshalling and convenience events so Delphi & Cbuilder applications can discover prompts, resources and tools exposed by any MCP-compliant server.
The client supports servers of the protocol version 2026-07-28 and of older versions, selecting the era automatically. See Protocol Versions, Multi Round-Trip Requests, Subscriptions, Tasks and Authorization.

Configuration

- **Drop the component:** place **TsgcWSAPIClient_MCP** on a form/data-module.
- **Point to an MCP endpoint:** set **MCPOptions.HttpOptions.URL** to the server HTTP URL that exposes the protocol (for example `https://localhost:5001/mcp`). Configure TLS and logging through the nested **TLSOptions** and **LogOptions** objects if needed.
- **Or connect over stdio:** set **MCPOptions.Transport** to `aimcptrStdio` and configure **MCPOptions.StdioOptions** (*Command*, *Arguments*, *WorkingDir*, *Environment*) with the server executable to launch. The client spawns the process and exchanges newline-delimited JSON-RPC over its standard input/output, with no HTTP URL required. This is how you connect to a local MCP server spawned as a subprocess.
- **Describe the client:** populate **MCPOptions.ClientInfo** with the product *Name*, human-readable *Title* and semantic *Version*. These values are forwarded during the **initialize** handshake.
- **Secure the session:** configure **MCPOptions.AuthenticationOptions** when the server expects an API key or a custom HTTP header. The credentials are attached to every HTTP request, including streaming upgrades.
- **Tune keep-alive:** use **MCPOptions.HeartBeat** to enable the automatic `ping` cycle and define the preferred interval (seconds). The component internally calls **Ping** when the internal keep-alive timer fires.
- **Handle events:** subscribe to the events described later in order to log responses, accept/reject sessions or post-process payloads returned by the server.

Reference

- Properties — 5 properties available on the component (2 published, 3 public).
- Methods — 18 public methods.
- Events — 22 events you can handle.

MCP Client | Protocol Versions

TsgcWSAPIClient_MCP talks to servers of both MCP eras. Servers that implement **2025-11-25** or older start with `initialize` and keep a session (`Mcp-Session-Id`). Servers that implement **2026-07-28** are stateless: there is no `initialize` and no session, every request carries the protocol version, the client information and the client capabilities in `_meta`, and the client learns the server features with `server/discover`.

Choosing the era

The property MCPOptions.ProtocolEra selects how the client connects when you call **Initialize**.

- **aimcpeAuto** (default): the client sends a `server/discover` probe. If the server answers, the modern era is used, otherwise the client falls back to the legacy `initialize` handshake.
- **aimcpeModern**: always uses the protocol version 2026-07-28.
- **aimcpeLegacy**: always uses `initialize` and a session, exactly as in previous releases.

Fallback rules (Auto)

- **HTTP transports**: the client falls back to legacy when the probe returns HTTP 400, 404 or 405, or one of the JSON-RPC errors `-32020`, `-32021`, `-32022` or `-32601`.
- **stdio transport**: the client falls back to legacy on `-32020`, `-32021` or `-32022`, and when a legacy server answers `-32601`. If the process does not answer the probe in time (the lower of 5 seconds and **StdioOptions.Timeout**) the client uses legacy too.
- **-32022 Unsupported protocol version**: the client reads the versions supported by the server from the error data and picks the highest version both sides support. It retries once in the modern era or falls back to legacy.
- Other failures, like HTTP 401, 403 or 5xx, never trigger a fallback. They are reported through **OnError** with the text `MCP: HTTP <code>` (followed by up to 200 characters of the body) and **Initialize** returns False.

The era detected for an endpoint is cached, so the next **Initialize** against the same URL or command does not repeat the probe.

Negotiated values

After **Initialize** the following public properties describe the connection:

- NegotiatedEra: `aimcpeModern` or `aimcpeLegacy`.
- NegotiatedProtocolVersion: the protocol version in use, for example `2026-07-28` or `2025-11-25`.
- ServerDiscover: the `server/discover` result (supported versions, capabilities and instructions). It is nil in the legacy era.

Discovery

In the modern era the component raises OnMCPDiscover with the `server/discover` result during **Initialize**. Set **Accept** to False to reject the server, then **Initialize** returns False. **OnMCPInitialize** is only raised in the legacy era. Call the method Discover at any time to read the server features again.

What changes in the modern era

- Every request includes `_meta` with `protocolVersion`, `clientInfo` and `clientCapabilities`.
- HTTP requests carry the headers `MCP-Protocol-Version`, `Mcp-Method` and `Mcp-Name` (base64 encoded when the value has characters that are not allowed in a header).
- There is no session id, no `notifications/initialized` and no GET stream. The heartbeat `ping` is not used.
- If a request fails with `-32022` the client retries it once with a version the server supports.

- Responses sent as `text/event-stream` are parsed, so log and progress notifications that come before the result are processed.
- Notifications are received through subscriptions instead of the session stream.

x-mcp-header parameters

A modern server can mark tool arguments with `x-mcp-header` in the input schema. The client reads these declarations when it receives the `tools/list` result and, on every `tools/call` of that tool, sends the argument value in a `Mcp-Param-<Name>` HTTP header. Integers are sent as numbers and booleans in lowercase. Tools with an invalid declaration are removed from the list and reported through **OnError** with the text `MCP tool "<name>" ignored: <reason>`.

Sample code

Connect with automatic era detection, log the negotiated version and reject servers that do not publish tools.

MCP Client | Subscriptions

In the MCP protocol version **2026-07-28** there is no session stream, so a client that wants to receive notifications opens a subscription with `subscriptions/listen`. The server keeps the response open and sends the notifications the client asked for: tools, prompts or resources list changes and updates of specific resources. For servers of the legacy era keep using SubscribeResource; the subscription methods below raise an exception in the legacy era.

Opening a subscription

Call SubscriptionsListen with the notifications you want and, optionally, the list of resource URIs to watch. The method returns the subscription id.

- **HTTP**: each subscription runs in its own background thread that keeps a POST open with `Accept: application/json, text/event-stream`.
- **stdio**: the subscription shares the pipe of the server process.

Events

- OnMCPSubscriptionAcknowledged: the server accepted the subscription. *aNotifications* is the raw JSON of the notifications the server honors, which can be a subset of the requested ones.
- OnMCPListChanged: a `list_changed` notification arrived. *aKind* is `aimcplckTools`, `aimcplckPrompts` or `aimcplckResources`; call ListTools, ListPrompts or ListResources to refresh the catalogue.
- **OnMCPResourcesUpdated**: a watched resource changed (`notifications/resources/updated`); read *aNotification.Uri* and call RequestResource. This event is published, assign it in the Object Inspector or in code. See OnMCPResourcesUpdated.
- OnMCPSubscriptionClosed: the subscription ended. *aGraceful* is True when the server closed it with a final result.

Closing a subscription

Call SubscriptionsCancel with the subscription id. With HTTP the stream is closed; with stdio the client sends `notifications/cancelled`. All subscriptions are cancelled automatically when **Initialize** is called again and when the component is destroyed.

Threads

With the HTTP transports the subscription events are raised in the background thread of the subscription, not in the main thread. Do not access visual controls directly from the handler, use `TThread.Queue` or `TThread.Synchronize`. With the stdio transport the events are synchronized with the main thread.
The server side is described in MCP Server | Subscriptions.

Sample code

MCP Client | Tasks

The tasks extension (`io.modelcontextprotocol/tasks`) of the MCP protocol version **2026-07-28** lets a server run a long `tools/call` in the background. Instead of the tool result, the server returns a task handle, and the client asks for the task status until the task finishes. The methods of this topic raise an exception in the legacy era.

Enabling tasks

Set **MCPOptions.Tasks.Enabled** to True. The client then declares the extension in `_meta.clientCapabilities` of every request, and the server decides for each `tools/call` whether it runs as a task.

Property	Description
Enabled	Declares the tasks extension. Default False.
AutoPoll	When True, a background thread polls `tasks/get` until the task reaches a terminal status. Default True.

Task lifecycle

- When a `tools/call` returns `resultType` **task**, OnMCPTaskCreated is raised first. Then **OnMCPResponse-Tool** is raised as usual, with the raw task handle JSON in *aResponse.TaskResult*.
- With **AutoPoll** the client calls `tasks/get` every `pollIntervalMs` milliseconds (1000 when the server sends none). OnMCPTaskStatus is raised every time the status, the status message or the requested inputs change.
- When the task reaches `completed`, `failed` or `cancelled`, OnMCPTaskCompleted is raised and polling stops. Read the tool result from *aTask.Result* or the error from *aTask.Error*.
- If the status is `input_required`, the inputs are answered like a Multi Round-Trip Request (legacy handlers first, then **OnMCPInputRequired** with *aMethod* `tasks/update`) and sent with `tasks/update`.
- If the server no longer knows the task, it is reported as failed. If the task time to live passes, the error is reported through **OnError** and polling stops.

Manual control

- TasksGet: requests the status of a task, raises OnMCPTaskStatus and returns the status.
- TasksUpdate: sends the answers of the inputs requested by a task.
- TasksCancel: asks the server to cancel a task and stops its poller.

TsgcAI_MCP_Client_Task

Property	Description
TaskId	Id of the task assigned by the server.
ToolName	Name of the tool that created the task.

Status	`aimcptskWorking`, `aimcptskInputRequired`, `aimcptskCompleted`, `aimcptskFailed` or `aimcptskCancelled`.
StatusMessage	Optional message sent by the server with the status.
CreatedAt, LastUpdatedAt	ISO 8601 timestamps sent by the server.
TTLMs	Time to live in milliseconds, -1 when unlimited.
PollIntervalMs	Poll interval suggested by the server.
Result	Tool result as raw JSON when the task is completed.
Error	Error as raw JSON when the task failed.
InputRequests	Inputs requested by the server as raw JSON.
Text	Raw JSON of the last task message.
IsTerminal	True when the status is completed, failed or cancelled.

Threads

The events raised by the poller (OnMCPTaskStatus and OnMCPTaskCompleted) run in the background thread of the task. Use `TThread.Queue` or `TThread.Synchronize` to update visual controls. Pollers are stopped when **Initialize** is called again and when the component is destroyed.
The server side is described in MCP Server | Tasks.

Sample code

MCP Client | Tools

The TsgcWSAPIClient_MCP component exposes the JSON-RPC calls defined by the Model Context Protocol (MCP) to enumerate tools that the server offers and to invoke a tool. Each request must be sent after Initialize succeeds because every method validates that the session is already in the *initialized* state.

ListTools request

Call **ListTools** to retrieve the registry of tools that the MCP server currently exposes. The component automatically builds the tools/list JSON-RPC request and raises the **OnMCPListTools** event with the parsed response.

- **Request:** ListTools (no parameters).
- **Event:** OnMCPListTools(Sender, Request, Response).
- **Response:** TsgcAI_MCP_Response_ToolsList with a Tools queue. Each item is a TsgcAI_MCP_Tool exposing the tool Name, Title, Description, and its InputSchema (JSON schema type plus named properties).

Use the queue's Count and Item[Index] helpers or GetTool to inspect the catalog.

ToolsCall request

Call **RequestTool**(Name, Arguments) to invoke one of the published tools. The component copies the JSON arguments into the request payload and raises **OnMCPResponseTool** after the response is parsed.

- **Request:** RequestTool with the tool **Name** and optional JSON **Arguments**.
- **Event:** OnMCPResponseTool(Sender, Request, Response).
- **Response:** TsgcAI_MCP_Response_ToolsCall. Inspect Result.IsError, Result.StructuredContent, and iterate Result.Content for the structured output fragments. Tool responses can contain text, images, audio blobs, embedded resources, or resource links.

Sample code

The following snippets show how to request the tool inventory, execute a tool, and read the results.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPListTools = MCPListTools;
    MCP->OnMCPResponseTool = MCPResponseTool;
}
void __fastcall TForm1::LoadTools()
{
    MCP->ListTools();
}
void __fastcall TForm1::MCPListTools(TObject *Sender,
    const TsgcAI_MCP_Request_ToolsList &ARequest,
    const TsgcAI_MCP_Response_ToolsList &AResponse)
{
    for (int LIndex = 0; LIndex < AResponse.Tools->Count; LIndex++)
    {
        TsgcAI_MCP_Tool *LTool =
            static_cast<TsgcAI_MCP_Tool*>(AResponse.Tools->Item[LIndex]);
        Memo1->Lines->Add(Format("%s (%s)", ARRAYOFCONST((LTool->Name, LTool->Description))));
    }
}
void __fastcall TForm1::CallTemperatureTool()
{
    TsgcJSON *LArgs = new TsgcJSON(nullptr);
    try
    {
        LArgs->AddPair("city", "Barcelona");
        MCP->RequestTool("GetTemperature", LArgs);
    }
    __finally
    {
        delete LArgs;
    }
}
```

```
void __fastcall TForm1::MCPResponseTool(TObject *Sender,
    const TsgcAI_MCP_Request_ToolsCall &ARequest,
    const TsgcAI_MCP_Response_ToolsCall &AResponse)
{
    if (AResponse.Result->IsError)
        Memo1->Lines->Add("Tool returned an error payload");
    for (int LIndex = 0; LIndex < AResponse.Result->Content->Count; LIndex++)
    {
        TsgcAI_MCP_Response_Result_Content *LContent =
            static_cast<TsgcAI_MCP_Response_Result_Content*>(
                AResponse.Result->Content->Item[LIndex]);
        Memo1->Lines->Add(LContent->Write());
    }
}
```

MCP Client | Prompts

The MCP Client component offers two JSON-RPC methods that revolve around server-provided prompts: listing the available prompt templates and requesting a prompt execution. Make sure the client has been initialized before issuing these calls.

PromptsList request

ListPrompts requests the current prompt catalog. When the response is received the component fires **OnMCPListPrompts**.

- **Request:** ListPrompts (no parameters).
- **Event:** OnMCPListPrompts(Sender, Request, Response).
- **Response:** TsgcAI_MCP_Response_PromptsList exposing a Prompts queue. Each TsgcAI_MCP_Prompt contains its Name, Title, Description, and Arguments collection (argument name, description, and required flag).

PromptsGet request

RequestPrompt(Name, Arguments) executes a prompt template with the supplied JSON arguments. The component raises **OnMCPResponsePrompt** when the server response is ready.

- **Request:** RequestPrompt with the prompt **Name** and optional JSON **Arguments**.
- **Event:** OnMCPResponsePrompt(Sender, Request, Response).
- **Response:** TsgcAI_MCP_Response_PromptsGet. Inspect Result.Description for a summary and iterate Result.Messages to read each TsgcAI_MCP_Response_PromptsGet_Result_Message. Messages are polymorphic—use the specific descendants (..._Text, ..._Image, ..._Audio, ..._EmbeddedResource) to access the underlying content and annotations.

Sample code

The example below loads the prompt list, invokes a prompt, and prints the returned messages.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPListPrompts = MCPListPrompts;
    MCP->OnMCPResponsePrompt = MCPResponsePrompt;
}
void __fastcall TForm1::LoadPrompts()
{
    MCP->ListPrompts();
}
void __fastcall TForm1::MCPListPrompts(TObject *Sender,
    const TsgcAI_MCP_Request_PromptsList &ARequest,
    const TsgcAI_MCP_Response_PromptsList &AResponse)
{
    for (int LIndex = 0; LIndex < AResponse.Prompts->Count; LIndex++)
    {
        TsgcAI_MCP_Prompt *LPrompt = (TsgcAI_MCP_Prompt*)AResponse.Prompts->Item[LIndex];
        Memo1->Lines->Add(Format("%s: %s", ARRAYOFCONST((LPrompt->Name, LPrompt->Description))));
    }
}
void __fastcall TForm1::ExecutePrompt()
{
    TsgcJSON *LArgs = new TsgcJSON(nullptr);
    try
    {
        LArgs->AddPair("code", "ShowMessage('Hello World')");
        MCP->RequestPrompt("CodeReview", LArgs);
    }
    __finally
    {
        delete LArgs;
    }
}
```

```
void __fastcall TForm1::MCPResponsePrompt(TObject *Sender,
    const TsgcAI_MCP_Request_PromptsGet &ARequest,
    const TsgcAI_MCP_Response_PromptsGet &AResponse)
{
    Memo1->Lines->Add("Prompt description: " + AResponse.Result->Description);
    for (int LIndex = 0; LIndex < AResponse.Result->Messages->Count; LIndex++)
    {
        TsgcAI_MCP_Response_PromptsGet_Result_Message *LMessage =
            (TsgcAI_MCP_Response_PromptsGet_Result_Message*)AResponse.Result->Messages->Item[LIndex];
        if (dynamic_cast<TsgcAI_MCP_Response_PromptsGet_Result_Message_Text*>(LMessage))
        {
            auto *MsgText = (TsgcAI_MCP_Response_PromptsGet_Result_Message_Text*)LMessage;
            Memo1->Lines->Add(LMessage->Role + ": " + MsgText->Content.Text);
        }
        else
        {
            Memo1->Lines->Add(LMessage->Role + ": " + LMessage->Write());
        }
    }
}
```

MCP Client | Resources

The MCP resource APIs let the client discover server-side artifacts and retrieve their contents through the **TsgcWSAPIClient_MCP** component once the session has been initialized.

ResourcesList request

Use **ListResources** to obtain the resource catalog. The component fires **OnMCPListResources** after parsing the JSON-RPC reply.

- **Request:** ListResources (no parameters).
- **Event:** OnMCPListResources(Sender, Request, Response).
- **Response:** TsgcAI_MCP_Response_ResourcesList exposing a Resources queue. Every TsgcAI_MCP_Resource entry includes the resource Uri, Name, Title, Description, and MimeType.

ResourcesRead request

Call **RequestResource**(Uri) to download a specific resource. The component raises **OnMCPResponseResource** with the decoded payload.

- **Request:** RequestResource with the target **Uri**.
- **Event:** OnMCPResponseResource(Sender, Request, Response).
- **Response:** TsgcAI_MCP_Response_ResourcesRead. Read Result.Contents to access each TsgcAI_MCP_Response_Resources_ReadResult_Content block which contains the Uri, Name, Title, Mime-Type, plus either Text or Blob data.

Sample code

This snippet lists the resources and retrieves a single file.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPListResources = MCPListResources;
    MCP->OnMCPResponseResource = MCPResponseResource;
}
void __fastcall TForm1::LoadResources()
{
    MCP->ListResources();
}
void __fastcall TForm1::MCPListResources(TObject *Sender,
    const TsgcAI_MCP_Request_ResourcesList &ARequest,
    const TsgcAI_MCP_Response_ResourcesList &AResponse)
{
    for (int LIndex = 0; LIndex < AResponse.Resources->Count; LIndex++)
    {
        TsgcAI_MCP_Resource *LResource =
            static_cast<TsgcAI_MCP_Resource*>(AResponse.Resources->Item[LIndex]);
        Memo1->Lines->Add(Format("%s -> %s", ARRAYOFCONST((LResource->Name, LResource->Uri))));
    }
}
void __fastcall TForm1::FetchResource(const String AUri)
{
    MCP->RequestResource(AUri);
}
void __fastcall TForm1::MCPResponseResource(TObject *Sender,
    const TsgcAI_MCP_Request_ResourcesRead &ARequest,
    const TsgcAI_MCP_Response_ResourcesRead &AResponse)
{
    for (int LIndex = 0; LIndex < AResponse.Result->Contents->Count; LIndex++)
    {
        TsgcAI_MCP_Response_Resources_ReadResult_Content *LContent =
            static_cast<TsgcAI_MCP_Response_Resources_ReadResult_Content*>(
                AResponse.Result->Contents->Item[LIndex]);
        if (LContent->Text != "")
            Memo1->Lines->Add(LContent->Uri + ": " + LContent->Text);
        else
```

```
            Memo1->Lines->Add(LContent->Uri + " (binary, mime=" + LContent->MimeType + ")");
    }
}
```

MCP Client | Roots

The MCP specification introduces the **roots** capability so that a client can describe the filesystem locations it is willing to expose to a server. Each root is a file:// URI accompanied by an optional display name. Servers query the catalogue with roots/list and, when supported, receive a notifications/roots/list_changed message whenever the selection changes. This handshake lets tooling respect project boundaries, workspaces and multi-repository layouts while keeping the user in control of what the server can access.

Understanding roots

Roots act as guardrails that scope server-side features. The client announces support during initialize by enabling the roots capability and optionally the listChanged flag so that dynamic updates can be pushed. A typical payload returned from roots/list contains one or more entries, each with Uri and Name, representing workspaces, repositories or other curated folders.

Handling roots/list requests

TsgcWSAPIClient_MCP parses incoming JSON-RPC messages and raises **OnMCPListRoots** whenever a server requests the current catalogue. Use the event to synchronise whatever workspace model your application maintains with the protocol response.

- **Request:** TsgcAI_MCP_Request_RootsList. The message does not include parameters but provides the JSON-RPC identifier you must mirror in the response.
- **Event:** OnMCPListRoots(Sender, Request, Response). This is triggered after the component validates the session.
- **Response:** TsgcAI_MCP_Response_RootsList. Call Roots.AddRoot(Uri, Name) (or build TsgcAI_MCP_Root instances manually) to describe each accessible location. The component serialises the populated object once your handler returns, so make sure the list reflects the current state. Use Response.Write for diagnostics or logging.

When your application allows users to add or remove workspaces at runtime, remember to raise the corresponding notifications/roots/list_changed notification so that servers refresh their cached catalogue.

Protocol eras

Legacy era (2025-11-25 and older): when the server sends roots/list, the component raises **OnMCPListRoots** and sends the reply to the server automatically when the handler returns, as a line on stdio or as an HTTP POST to the endpoint with the session and authorization headers. If **OnMCPListRoots** is not assigned, the client answers with the JSON-RPC error -32601 (Method not found). The roots capability is declared in initialize only when **OnMCPListRoots** is assigned or MCPOptions.MRTR.Roots is True.
Protocol version 2026-07-28: the server cannot send this request, it asks for the input with an input_required result instead. The component calls the same **OnMCPListRoots** handler to answer it automatically and repeats the original request, so the same code works in both eras. See Multi Round-Trip Requests.

Sample code

The following snippets publish two roots when the server asks for them and log the JSON payload that will be sent back.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPListRoots = MCPListRoots;
}
void __fastcall TForm1::MCPListRoots(TObject *Sender,
    const TsgcAI_MCP_Request_RootsList *ARequest,
    const TsgcAI_MCP_Response_RootsList *AResponse)
{
    AResponse->Roots->Clear();
```

```
    AResponse->Roots->AddRoot("file:///c:/projects/app", "Application Workspace");
    AResponse->Roots->AddRoot("file:///c:/projects/libs", "Shared Libraries");
    MemoLog->Lines->Add("Roots response: " + AResponse->Write());
}
```

MCP Client | Sampling

Sampling lets MCP servers delegate language-model generations to the client without sharing provider credentials. A server issues sampling/createMessage requests that encapsulate the conversation state, model preferences and optional constraints such as maxTokens. The client can involve a human reviewer, enforce guard rails and ultimately return a response that includes the selected model, generated content and stop reason.

Understanding sampling

When the sampling capability is advertised during initialize, servers are free to ask for generations mid-workflow. Requests bundle message arrays (text, image or audio blocks), a systemPrompt, optional token limits and a ModelPreferences structure that balances cost, speed and intelligence via numeric priorities and hint strings. The client is responsible for mapping these hints to available models, prompting the user when necessary and filtering unsafe completions before replying.

Handling sampling/createMessage requests

TsgcWSAPIClient_MCP surfaces server initiated generations through **OnMCPSamplingCreateMessage**. Your handler should examine the provided context, route the call to the desired model (or human reviewer) and populate the response payload.

- **Request:** TsgcAI_MCP_Request_SamplingCreateMessage. Inspect Params.Messages to replay the dialogue, Params.ModelPreferences for hint prioritisation and Params.SystemPrompt for guardrails.
- **Event:** OnMCPSamplingCreateMessage(Sender, Request, Response). Raised after the component validates the MCP session and deserialises the request.
- **Response:** TsgcAI_MCP_Response_SamplingCreateMessage. Provide the assistant Role, assign a content object (text, image, audio or resource link), and optionally record the concrete Model and StopReason. Call Response.Write to preview the JSON payload before it is dispatched.

Protocol eras

Legacy era (2025-11-25 and older): when the server sends sampling/createMessage, the component raises **OnMCPSamplingCreateMessage** and sends the reply to the server automatically when the handler returns, as a line on stdio or as an HTTP POST to the endpoint with the session and authorization headers. If **OnMCPSamplingCreateMessage** is not assigned, the client answers with the JSON-RPC error -32601 (Method not found). The sampling capability is declared in initialize only when **OnMCPSamplingCreateMessage** is assigned or MCPOptions.MRTR.Sampling is True.
Protocol version 2026-07-28: the server cannot send this request, it asks for the input with an input_required result instead. The component calls the same **OnMCPSamplingCreateMessage** handler to answer it automatically and repeats the original request, so the same code works in both eras. See Multi Round-Trip Requests.

Sample code

The snippets below log the incoming message count, generate a fixed reply and annotate the model and stop reason. Substitute the hard-coded content with a real model invocation or a human approval loop in production.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPSamplingCreateMessage = MCPSamplingCreateMessage;
}
void __fastcall TForm1::MCPSamplingCreateMessage(TObject *Sender,
    const TsgcAI_MCP_Request_SamplingCreateMessage *ARequest,
    const TsgcAI_MCP_Response_SamplingCreateMessage *AResponse)
{
    MemoLog->Lines->Add(Format("Sampling request with %d messages", ARRAYOFCONST((ARequest->Params->Messages->Cou
    AResponse->Role = "assistant";
    auto *text = new TsgcAI_MCP_Response_Result_Content_Text();
    text->Text = "The capital of France is Paris.";
    AResponse->Content = text;
```

```
    AResponse->Model = "claude-3-sonnet-20240307";
    AResponse->StopReason = "endTurn";
    MemoLog->Lines->Add("Sampling response: " + AResponse->Write());
}
```

MCP Client | Elicitation

Elicitation enables a server to request structured input from the user through the MCP client. The request carries a human readable message and a constrained JSON Schema that defines the expected fields. Clients keep control of the interaction, validate the supplied data and return an action that indicates whether the user accepted, declined or cancelled the prompt.

Understanding elicitation

The elicitation capability is negotiated during initialize. Once enabled, servers may send elicitation/create calls that embed a flat object schema limited to primitive types (string, number/integer, boolean and enum with optional display names). Each property can specify display metadata, length or range constraints and supported formats (email, uri, date, date-time). Users should always be informed about who is asking for the information, be given a chance to edit the payload and be able to refuse or cancel the request altogether.

Handling elicitation/create requests

TsgcWSAPIClient_MCP raises **OnMCPElicitationCreate** whenever the server demands additional input. The event gives you access to the prompt message, the requested schema and an empty response object that you can populate based on user decisions.

- **Request:** TsgcAI_MCP_Request_ElicitationCreate. Inspect Params.Message to display a prompt and Params.RequestedSchema.Properties to build UI controls or validations.
- **Event:** OnMCPElicitationCreate(Sender, Request, Response). Triggered after the client deserialises the JSON-RPC payload.
- **Response:** TsgcAI_MCP_Response_ElicitationCreate. Set Action to accept, decline or cancel. When accepting, write the collected values into Content (a TsgcJSON helper) so they match the requested schema. Use Response.Write to inspect the final JSON document.

Protocol eras

Legacy era (2025-11-25 and older): when the server sends elicitation/create, the component raises **OnMCPElicitationCreate** and sends the reply to the server automatically when the handler returns, as a line on stdio or as an HTTP POST to the endpoint with the session and authorization headers. If **OnMCPElicitationCreate** is not assigned, the client answers with the JSON-RPC error -32601 (Method not found). The elicitation capability is declared in initialize only when **OnMCPElicitationCreate** is assigned or MCPOptions.MRTR.Elicitation is True.
Protocol version 2026-07-28: the server cannot send this request, it asks for the input with an input_required result instead. The component calls the same **OnMCPElicitationCreate** handler to answer it automatically and repeats the original request, so the same code works in both eras. See Multi Round-Trip Requests.

Sample code

The example below prompts the user for a GitHub handle. If the operator supplies a value, the response is accepted and the field is returned; otherwise the request is declined.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPElicitationCreate = MCPElicitationCreate;
}
void __fastcall TForm1::MCPElicitationCreate(TObject *Sender,
    const TsgcAI_MCP_Request_ElicitationCreate *ARequest,
    const TsgcAI_MCP_Response_ElicitationCreate *AResponse)
{
    MemoLog->Lines->Add("Elicitation prompt: " + ARequest->Params->Message);
    String gitHub = InputBox("GitHub", "Provide your GitHub username", "");
    if (!gitHub.IsEmpty())
    {
        AResponse->Action = "accept";
        AResponse->Content->Clear();
        AResponse->Content->AddPair("name", gitHub);
```

```
    }
    else
    {
        AResponse->Action = "decline";
    }
    MemoLog->Lines->Add("Elicitation response: " + AResponse->Write());
}
```

MCP Client | Multi Round-Trip Requests

In the MCP protocol version **2026-07-28** a server cannot send requests to the client (roots, sampling or elicitation) in the middle of a tool call, because there is no session. The server answers `tools/call`, `prompts/get` or `resources/read` with `resultType` **input_required** and a list of `inputRequests`. The client collects the answers and repeats the same request with a new id, the answers in `inputResponses` and the exact `requestState` received from the server. This is a Multi Round-Trip Request (MRTR). **TsgcWSAPIClient_MCP** runs the whole loop for you inside **Request-Tool**, **RequestPrompt** and **RequestResource**.

How the inputs are answered

- **Automatic answers**: each input of type `roots/list`, `sampling/createMessage` or `elicitation/create` is passed to the handler you already use for legacy servers: OnMCPListRoots, OnMCPSamplingCreateMessage or OnMCPElicitationCreate. The same code works in both eras.
- **OnMCPInputRequired**: then OnMCPInputRequired is raised with the original method (`tools/call`, `prompts/get`, `resources/read`, or `tasks/update` for a task) and the list of inputs. **Accept** is True when every input already has an answer. Fill the missing answers and leave **Accept** True to send the next round, or set it to False to stop.

When the loop stops without an answer, the caller receives the `input_required` response (its **ResultType** is `aimcprtInputRequired`) and no error is raised. If the server asks for input more than **MCPOptions.MRTR.MaxRounds** times, the loop stops and the error is reported through **OnError**.

MCPOptions.MRTR

Property	Description
MaxRounds	Maximum number of rounds for one request. Default 10.
Elicitation	Declares the `elicitation` client capability. Default False.
Sampling	Declares the `sampling` client capability. Default False.
Roots	Declares the `roots` client capability. Default False.

The client declares a capability in `_meta.clientCapabilities` (and in the legacy `initialize`) when the matching handler (OnMCPElicitationCreate, OnMCPSamplingCreateMessage or OnMCPListRoots) is assigned or the MRTR option is True. Enable the options when you answer the inputs only from **OnMCPInputRequired**, because a server does not ask for inputs the client did not declare.

TsgcAI_MCP_Client_InputRequests

Member	Description
Count	Number of inputs requested by the server.
Keys[i]	Key of the input, used to match the answer.
Methods[i]	Method of the input, for example `elicitation/create` .

Params[i]	Params of the input as raw JSON.
Responses[i]	Answer already assigned (raw JSON), empty when there is none.
SetResponse(aKey, aResultJSON)	Sets the answer of an input as a raw JSON result.
SetElicitationAccept(aKey, aContentJSON)	Answers an elicitation with `action` accept and the form content (JSON object).
SetElicitationDecline(aKey)	Answers an elicitation with `action` decline.
SetRoots(aKey, aRootsJSON)	Answers a roots request with a JSON array of roots.
IndexOfKey(aKey)	Returns the index of a key or -1.
IsCompleted	True when every input has an answer.

The server side of this feature is described in MCP Server | Multi Round-Trip Requests.

Sample code

Answer an elicitation that asks for the user name and publish one root. Declare both capabilities because no legacy handler is assigned.

MCP Client | Authentication (OAuth2)

The TsgcWSAPIClient_MCP component can authenticate against an MCP server with an OAuth2 bearer token. When MCPOptions.AuthenticationOptions.OAuth2.Enabled is set, the client obtains an access token through the OAuth2 client and sends an Authorization: Bearer <token> header on every MCP HTTP request.
The default grant is the Identity Assertion Authorization Grant, which propagates the user identity across security domains without an interactive login. You can select any other supported grant through the GrantType property.
To let the client discover the authorization server, register itself and log the user in automatically, as defined by the MCP authorization specification, use MCPOptions.Authorization instead.

Precedence and caching

OAuth2 authentication takes precedence over ApiKey. When OAuth2.Enabled is True, the client sends the bearer token and ignores the API key, even if ApiKey.Enabled is also set.
The token is cached after it is first obtained and reused on subsequent requests until it is close to expiry, at which point the client refreshes it automatically. This avoids a round-trip to the authorization server on every MCP call.

Options

The MCPOptions.AuthenticationOptions.OAuth2 object exposes the following properties.

Property	Description
Enabled	When True, the client obtains a bearer token through OAuth2 and sends it on every MCP HTTP request. Takes precedence over ApiKey.
TokenURL	The token endpoint of the resource authorization server where the access token is obtained.
ClientId	The client identifier used to authenticate at the resource authorization server.
ClientSecret	The client secret used to authenticate at the resource authorization server.
Scope	The scope requested for the access token.
GrantType	The OAuth2 grant used to obtain the token. Default auth2IdentityAssertion.
IdentityAssertionOptions	Settings for the Identity Assertion grant (SubjectToken, Audience, RequestingPartyTokenURL, and the rest). These mirror the fields of the OAuth2 client. See the Identity Assertion Authorization Grant topic.

Sample code

The following snippet configures the MCP client to authenticate with the Identity Assertion grant against an MCP server URL, then calls Initialize to open the session.

```
MCP->MCPOptions->HttpOptions->URL = "https://mcp.partner.com/mcp";
MCP->MCPOptions->Transport = aimcptrHttpStreamable;
MCP->MCPOptions->AuthenticationOptions->OAuth2->Enabled = true;
MCP->MCPOptions->AuthenticationOptions->OAuth2->GrantType = auth2IdentityAssertion;
MCP->MCPOptions->AuthenticationOptions->OAuth2->TokenURL = "https://auth.partner.com/oauth2/token";
MCP->MCPOptions->AuthenticationOptions->OAuth2->ClientId = "your-client-id";
```

```
MCP->MCPOptions->AuthenticationOptions->OAuth2->ClientSecret = "your-client-secret";
MCP->MCPOptions->AuthenticationOptions->OAuth2->Scope = "mcp.read mcp.write";
MCP->MCPOptions->AuthenticationOptions->OAuth2->IdentityAssertionOptions->RequestingPartyTokenURL = "https://idp.
MCP->MCPOptions->AuthenticationOptions->OAuth2->IdentityAssertionOptions->SubjectToken = "the-users-existing-acce
MCP->MCPOptions->AuthenticationOptions->OAuth2->IdentityAssertionOptions->Audience = "https://mcp.partner.com";
MCP->Initialize();
```

MCP Client | Authorization

The MCP authorization specification (protocol version **2026-07-28**) describes how a client gets an access token for a protected MCP server without any manual configuration. When **MCPOptions.Authorization.Enabled** is True and the server answers a request with HTTP 401 or 403, **TsgcWSAPIClient_MCP** discovers the authorization server, registers itself, opens the browser for the user login, gets the token and repeats the request with an `Authorization: Bearer` header. It works in both protocol eras over the HTTP transports.
This feature is different from MCPOptions.AuthenticationOptions.OAuth2, where you configure a token endpoint and a grant yourself. The authorization members are compiled only when OAuth2 support is enabled in the library (define `SGC_OAUTH`).

Flow

- **Protected resource metadata**: the client reads `resource_metadata` from the `WWW-Authenticate` header, or tries the well-known path of the endpoint and then the root. When several authorization servers are listed, the one equal to **Issuer** is used, otherwise the first.
- **Authorization server metadata**: tried in this order: `oauth-authorization-server`, `openid-configuration` with the path inserted, and the path followed by `/.well-known/openid-configuration`. The `issuer` of the metadata must be identical to the discovered issuer.
- **Client registration**: a pre-registered **ClientId** is used first. Otherwise, if the server supports client id metadata documents, **ClientMetadataURL** (an https URL) is sent as the client id. Otherwise, when **AllowDynamicRegistration** is True, the client registers with Dynamic Client Registration, sending `application_type native` for a loopback **RedirectURL** and `web` for any other.
- **Authorization request**: PKCE with S256 and the `resource` parameter (RFC 8707) set to the canonical URL of the MCP server. The client starts a local HTTP listener on **RedirectURL** and raises **OnMCPAuthorizationURL** with the URL the user must open. It waits up to **Timeout** seconds for the redirect.
- **Issuer check**: the `iss` parameter of the redirect (RFC 9207) must match the issuer, otherwise the flow stops.
- **Token**: the code is exchanged for a token and the original request is repeated. A request is authorized at most twice.
- **Refresh**: a token that expires in less than 30 seconds is refreshed before use. A 401 from the same issuer tries the refresh token before a new login.
- **Step-up**: a 403 with `insufficient_scope` starts a new authorization that requests the union of the current and the required scopes.

Credentials are kept per issuer, so a pre-registered client bound to one issuer is never sent to a different authorization server. Every failure is reported through **OnError** with a text that starts with `MCP authorization:`.

MCPOptions.Authorization

Property	Description
Enabled	Enables the automatic authorization. Default False.
ClientId	Client id of a pre-registered client. Optional.
ClientSecret	Client secret of a pre-registered confidential client. Optional.
Issuer	Preferred authorization server when the resource lists several. Optional.
ClientMetadataURL	https URL of the client id metadata document of your application. Optional.
AllowDynamicRegistration	Allows Dynamic Client Registration when no other registration applies. Default True.

ClientName	Client name sent with Dynamic Client Registration.
RedirectURL	Redirect URL where the local listener waits for the authorization code. Default `http://127.0.0.1:33418/callback`.
Timeout	Seconds to wait for the user login. Default 300.
AccessToken	Read only. Bearer token currently in use, empty until authorized.

Events

- OnMCPAuthorizationURL: raised with the authorization URL. Set **Handled** to True if your application opens it (for example in an embedded browser); otherwise the default browser is opened.
- OnMCPAuthorizationCredentials: raised with *aSave* False the first time an issuer is used, so you can load stored credentials into *aCredentials*, and with *aSave* True after a registration, a new token or a refresh, so you can store them.

TsgcAI_MCP_Client_AuthorizationCredentials has the properties Issuer (read only), ClientId, ClientSecret, AccessToken, RefreshToken, ExpiresAt, Scope and Registration (`preregistered`, `cimd` or `dcr`).

Sample code

OpenAI

OpenAI is a private research laboratory that aims to develop and direct artificial intelligence (AI) in ways that benefit humanity as a whole. OpenAI has developed the following projects:

- **GPT-3:** This powerful language model serves as the basis for other OpenAI products. It analyzes human-generated text to learn to generate similar text on its own.
- **DALL-E and DALL-E 2:** These generative AI platforms can analyze text-based descriptions of images that users want them to produce and then generate those images exactly as described.
- **CLIP:** CLIP is a neural network that synthesizes visuals and text pertaining to them to predict the best possible captions that most accurately describe those visuals. Because of its ability to learn from more than one type of data (both images and text), it can be categorized as multimodal AI.
- **ChatGPT:** ChatGPT is currently the most advanced AI chatbot designed for generating humanlike text and producing answers to users' questions. Having been trained on large data sets, it can generate answers and responses the way a human would.
- **Codex:** Codex was trained on billions of lines of code in various programming languages to help software developers simplify coding processes. It's founded on GPT-3 technology, but instead of generating text, it generates code.
- **Whisper:** Whisper is an automatic speech recognition (ASR) tool. It has been trained on a multitude of audio data to recognize, transcribe, and translate speech in about 100 different languages, including technical language and different accents.

OpenAI API

The OpenAI API can be applied to virtually any task that involves understanding or generating natural language, code, or images. OpenAI offers a spectrum of models with different levels of power suitable for different tasks, as well as the ability to fine-tune your own custom models. These models can be used for everything from content generation to semantic search and classification.

Most common uses

- **Completion**
  - **OpenAI Completion Examples**
- **Chat**
  - **OpenAI Chat Examples**
- **Edit**
  - **OpenAI Edit Examples**
- **Audio**
  - **OpenAI Transcribe & Translate Examples**
- **Moderation**
  - **OpenAI Moderation Examples**
- **RealTime**
  - **OpenAI RealTime Examples**
- **Responses**
  - **OpenAI Responses Examples**
- **Speech**
  - **OpenAI Speech Examples**
- **Fine-Tuning**
  - **OpenAI Fine-Tuning Examples**
- **Batch**
  - **OpenAI Batch Examples**

- **Uploads**
  - **OpenAI Uploads Examples**

Configuration

OpenAI

The OpenAI API uses API keys for authentication. Visit your API Keys page to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code (browsers, apps). Production requests must be routed through your own backend server where your API key can be securely loaded from an environment variable or key management service.

This **API Key** must be configured in the **OpenAIOptions.ApiKey** property of the component. Optionally, for users who belong to multiple organizations, you can set your Organization in the property **OpenAIOptions.Organization** if your account belongs to an organization.

Once the API Key is configured, find below a list of available functions to interact with the OpenAI API.

Azure

The client supports Microsoft Azure OpenAI Services, so you can use your Azure account to interact with the Azure OpenAI API too. To configure the client to work with Azure, follow the steps below:

1. Configure the property **OpenAIOptions.Provider** = oapvAzure
2. Set the values of ResourceName and DeploymentId (these values can be located in your Azure Account)
    1. **OpenAIOptions.AzureOptions.ResourceName** = <your resource name>.
    2. **OpenAIOptions.AzureOptions.DeploymentId** = <your deployment id>.
3. Set the API Key of your Azure Account
    1. **OpenAIOptions.ApiKey** = <azure api key>.

Keep in mind that not all the OpenAI methods are supported by Azure, currently only the following methods are supported:

1. Completion
2. Chat Completion

Properties

OpenAIOptions

- **ApiKey:** The API key for authenticating with the OpenAI API.
- **Organization:** Optional organization ID for users belonging to multiple organizations.
- **Provider:** Select the provider: oapvOpenAI (default) or oapvAzure for Microsoft Azure OpenAI Services.
- **AzureOptions:** Configuration for Azure OpenAI Services.
    - **ResourceName:** The Azure resource name.
    - **DeploymentId:** The Azure deployment ID.
    - **APIVersion:** The Azure API version.
- **HttpOptions:** HTTP connection settings.
    - **ReadTimeout:** Timeout in milliseconds for reading HTTP responses. Default is 0 (no timeout).
- **LogOptions:** Logging configuration.
    - **Enabled:** When True, HTTP requests and responses are logged to a file.
    - **FileName:** The file path where log output is written.
- **RetryOptions:** Automatic retry configuration for failed API requests.
    - **Enabled:** When True, failed requests are automatically retried.
    - **Retries:** Maximum number of retry attempts.
    - **Wait:** Wait time in milliseconds between retry attempts.

Models

List and describe the various models available in the API.

- **GetModels:** Lists the currently available models, and provides basic information about each one such as the owner and availability.
- **GetModel:** Retrieves a model instance, providing basic information about the model such as the owner and permissioning.
    - **Model:** The ID of the model to use for this request

Completions

Given a prompt, the model will return one or more predicted completions, and can also return the probabilities of alternative tokens at each position.

- **CreateCompletion:** Creates a completion for the provided prompt and parameters
    - **Model:** ID of the model to use. You can use the List models API to see all of your available models, or see our Model overview for descriptions of them.
    - **Prompt:** The prompt to generate completions.

Chat

Given a chat conversation, the model will return a chat completion response.

- **Model:** ID of the model to use. Call GetModels to get a list of all models supported by the Chat API.
- **Message:** The message to generate chat completions for.
- **Role:** by default user, other options are: system, assistant.

Edits

Given a prompt and an instruction, the model will return an edited version of the prompt.

- **CreateEdit:** Creates a new edit for the provided input, instruction, and parameters.
    - **Model:** ID of the model to use. You can use the text-davinci-edit-001 or code-davinci-edit-001 model with this endpoint.
    - **Instruction:** The instruction that tells the model how to edit the prompt.
    - **Input:** (optional) The input text to use as a starting point for the edit.

Images

Given a prompt and/or an input image, the model will generate a new image.

- **CreateImage:** Creates an image given a prompt.
    - **Prompt:** A text description of the desired image(s). The maximum length is 1000 characters.
- **CreateImageEdit:** Creates an edited or extended image given an original image and a prompt.
    - **Image:** The image to edit. Must be a valid PNG file, less than 4MB, and square. If mask is not provided, image must have transparency, which will be used as the mask.
    - **Prompt:** A text description of the desired image(s). The maximum length is 1000 characters.
- **CreateImageVariations:** Creates a variation of a given image.
    - **Image:** The image to use as the basis for the variation(s). Must be a valid PNG file, less than 4MB, and square.

Embeddings

Get a vector representation of a given input that can be easily consumed by machine learning models and algorithms.

- **CreateEmbeddings:** Creates an embedding vector representing the input text.
  - **Model:** ID of the model to use.
  - **Input:** Input text to get embeddings for.

Audio

Turn Audio into Text.

- **CreateTranscriptionFromFile:** Transcribes audio into the input language from a filename
  - **Model:** ID of the model to use. Only whisper-1 is currently available.
  - **Filename:** The audio file to transcribe, in one of these formats: mp3, mp4, mpeg, mpga, m4a, wav, or webm.
- **CreateTranscription:** Records audio for X seconds and transcribes it.
  - **Model:** ID of the model to use. Only whisper-1 is currently available.
  - **Time:** time in milliseconds, by default 10 seconds.
- **CreateTranslationFromFile:** Translates audio into English.
  - **Model:** ID of the model to use. Only whisper-1 is currently available.
  - **Filename:** The audio file to translate, in one of these formats: mp3, mp4, mpeg, mpga, m4a, wav, or webm.
- **CreateTranslation:** Records audio for X seconds and translates it.
  - **Model:** ID of the model to use. Only whisper-1 is currently available.
  - **Time:** time in milliseconds, by default 10 seconds.

Files

Files are used to upload documents that can be used with features like Fine-tuning.

- **ListFiles:** Returns a list of files that belong to the user's organization.
- **UploadFile:** Upload a file that contains document(s) to be used across various endpoints/features. Currently, the size of all the files uploaded by one organization can be up to 1 GB.
  - **Filename:** Name of the JSON Lines file to be uploaded. If the purpose is set to "fine-tune", each line is a JSON record with "prompt" and "completion" fields representing your training examples.
  - **Purpose:** The intended purpose of the uploaded documents. Use "fine-tune" for Fine-tuning.
- **DeleteFile:** Delete a file.
  - **FileId:** The ID of the file to use for this request
- **RetrieveFile:** Returns information about a specific file.
  - **FileId:** The ID of the file to use for this request
- **RetrieveFileContent:** Returns the contents of the specified file
  - **FileId:** The ID of the file to use for this request.

Fine-Tunes

Manage fine-tuning jobs to tailor a model to your specific training data.

- **CreateFineTune:** Creates a job that fine-tunes a specified model from a given dataset. Response includes details of the enqueued job including job status and the name of the fine-tuned models once complete.
  - **TrainingFile:** The ID of an uploaded file that contains training data.
- **ListFineTunes:** List your organization's fine-tuning jobs
- **RetrieveFineTune:** Gets info about the fine-tune job.
  - **FineTuneId:** The ID of the fine-tune job
- **CancelFineTune:** Immediately cancel a fine-tune job.
  - **FineTuneId:** The ID of the fine-tune job
- **ListFineTuneEvents:** Get fine-grained status updates for a fine-tune job.
  - **FineTuneId:** The ID of the fine-tune job
- **DeleteFineTuneModel:** Delete a fine-tuned model. You must have the Owner role in your organization.
  - **Model:** The model to delete.

Moderations

Given an input text, outputs if the model classifies it as violating OpenAI's content policy.

- **CreateModeration:** Classifies if text violates OpenAI's Content Policy
  - **Input:** The input text to classify

RealTime

The OpenAI Realtime API enables low-latency, multimodal interactions including speech-to-speech conversational experiences and real-time transcription.

- **Transcription:** You can use the Realtime API for transcription-only use cases, either with input from a microphone or from a file. For example, you can use it to generate subtitles or transcripts in real-time. With the transcription-only mode, the model will not generate responses.

Assistants

Build AI assistants that can call models and use tools to perform tasks.

- **CreateAssistant:** Creates an assistant with a model and instructions.
- **ListAssistants:** Returns a list of assistants.
- **RetrieveAssistant:** Retrieves an assistant by ID.
- **ModifyAssistant:** Modifies an existing assistant.
- **DeleteAssistant:** Deletes an assistant.

Threads

Threads represent a conversation session. Messages are added to threads, which are then processed by runs.

- **CreateThread:** Creates a new thread.
- **RetrieveThread:** Retrieves a thread by ID.
- **ModifyThread:** Modifies a thread.
- **DeleteThread:** Deletes a thread.

Thread Messages

Messages are added to threads and contain the content of a conversation.

- **CreateMessage:** Creates a message within a thread.
  - **ThreadId:** The ID of the thread.
- **ListMessages:** Returns a list of messages for a given thread.
- **RetrieveMessage:** Retrieves a message by thread and message ID.
- **ModifyMessage:** Modifies a message.
- **DeleteMessage:** Deletes a message from a thread.

Runs

Runs represent an execution on a thread with an assistant. The assistant uses its configuration and the thread messages to perform tasks by calling models and tools.

- **CreateRun:** Creates a run for a thread with a specified assistant.
- **ListRuns:** Returns a list of runs belonging to a thread.
- **RetrieveRun:** Retrieves a run by thread and run ID.
- **ModifyRun:** Modifies a run.

- **SubmitToolOutputsToRun:** Submits tool outputs for a run that requires action.
- **CancelRun:** Cancels an in-progress run.

Run Steps

Run steps represent the individual steps taken during a run execution.

- **ListRunSteps:** Returns a list of run steps belonging to a run.
- **RetrieveRunSteps:** Retrieves a run step by thread, run, and step ID.

Vector Stores

Vector stores are used to store and search over files using embeddings for retrieval-augmented generation (RAG).

- **CreateVectorStore:** Creates a vector store.
- **ListVectorStores:** Returns a list of vector stores.
- **RetrieveVectorStore:** Retrieves a vector store by ID.
- **ModifyVectorStore:** Modifies a vector store.
- **DeleteVectorStore:** Deletes a vector store.

Vector Store Files

Manage files within vector stores.

- **CreateVectorStoreFile:** Attaches a file to a vector store.
- **ListVectorStoreFiles:** Returns a list of files in a vector store.
- **RetrieveVectorStoreFile:** Retrieves a vector store file.
- **DeleteVectorStoreFile:** Removes a file from a vector store.

Vector Store File Batches

Batch operations for adding files to vector stores.

- **CreateVectorStoreFileBatch:** Creates a batch to add multiple files to a vector store.
- **RetrieveVectorStoreFileBatch:** Retrieves a file batch by ID.
- **CancelVectorStoreFileBatch:** Cancels an in-progress file batch.
- **ListVectorStoreFilesBatch:** Lists files in a batch.

Speech

Generate spoken audio from text using text-to-speech models.

- **CreateSpeech:** Generates audio from the input text. Returns audio data as a string or writes to a response stream.
  - **Model:** The TTS model to use (e.g. tts-1, tts-1-hd).
  - **Input:** The text to generate audio for.
  - **Voice:** The voice to use (alloy, echo, fable, onyx, nova, shimmer).

Fine-Tuning Jobs

Manage fine-tuning jobs to create customized models. This is the newer fine-tuning API that replaces the legacy Fine-Tunes endpoint.

- **CreateFineTuningJob:** Creates a fine-tuning job from a training file.
- **ListFineTuningJobs:** Lists your organization's fine-tuning jobs.
- **RetrieveFineTuningJob:** Retrieves info about a fine-tuning job.
- **CancelFineTuningJob:** Cancels a fine-tuning job.
- **ListFineTuningJobEvents:** Lists status updates for a fine-tuning job.
- **ListFineTuningJobCheckpoints:** Lists checkpoints for a fine-tuning job.

Responses

Create and manage model responses. The Responses API supports multi-turn conversations, tool use, and structured outputs.

- **CreateResponse:** Creates a model response.

- **RetrieveResponse:** Retrieves a response by ID.
- **DeleteResponse:** Deletes a response.
- **CancelResponse:** Cancels an in-progress response.
- **ListInputItems:** Lists input items for a response.

Batches

Create and manage batch API requests for asynchronous processing at lower cost.

- **CreateBatch:** Creates a batch job.
- **RetrieveBatch:** Retrieves a batch by ID.
- **ListBatches:** Lists all batches.
- **CancelBatch:** Cancels an in-progress batch.

Uploads

Upload large files in parts. Useful for files that exceed the standard upload size limit.

- **CreateUpload:** Creates an upload session.
- **AddUploadPart:** Adds a part to an upload.
- **CompleteUpload:** Completes a multi-part upload and creates a file.
- **CancelUpload:** Cancels an in-progress upload.

TsgcHTTP_API_OpenAI

OpenAI REST API client covering Chat Completions, Images, Audio (TTS/STT), Embeddings, Files, Fine-tuning, Moderation, Assistants, Threads, Runs, Messages and Vector Stores.

Introduction

OpenAI REST API client covering Chat Completions, Images, Audio (TTS/STT), Embeddings, Files, Fine-tuning, Moderation, Assistants, Threads, Runs, Messages and Vector Stores.

Introduction

OpenAI REST API client covering Chat Completions, Images, Audio (TTS/STT), Embeddings, Files, Fine-tuning, Moderation, Assistants, Threads, Runs, Messages and Vector Stores.
Configure the **OpenAIOptions** property with your API key and optional HTTP/retry/log settings, then call the typed methods such as `CreateMessage`/`CreateChatCompletion` with a request object and receive a typed response class.
For background on the provider, see the OpenAI guide.

Reference

- Properties — 5 properties available on the component.
- Methods — 78 public methods.
- Events — 2 events you can handle.

OpenAI | Completion

Given a prompt, the model will return one or more predicted completions, and can also return the probabilities of alternative tokens at each position.

Simple Example

Use the text-davinci-003 model to get a predicted completion.

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";
ShowMessage(OpenAI->_CreateCompletion("text-davinci-003", "Say this is a test"));
```

Advanced Example

Use the text-davinci-003 model to get a predicted completion with more random output and generate 2 completions for each prompt.

```
TsgcHTTP_OpenAI_JSON *OpenAI = new TsgcHTTP_OpenAI_JSON(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

TsgcOpenAIClass_Request_Completion *oRequest = new TsgcOpenAIClass_Request_Completion();
try
{
  oRequest->Model = "text-davinci-003";
  oRequest->Prompt = "Say this is a test";
  oRequest->Temperature = 1;
  oRequest->N = 2;
  TsgcOpenAIClass_Response_Completion *oResponse = OpenAI->CreateCompletion(oRequest);

  if (Length(oResponse->Choices) > 0)
  {
    ShowMessage(oResponse->Choices[0]->Text);
  }
}
__finally
{
  delete oRequest;
  delete oResponse;
}
```

OpenAI | Chat

Given a chat conversation, the model will return a chat completion response.

Simple Example

Interact with ChatGPT by sending a Hello message.

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";
ShowMessage(OpenAI->_CreateChatCompletion("gpt-3.5-turbo", "Hello!"));
```

Advanced Example

Use the gpt-3-5 model to chat with more random output and generate 2 completions for each prompt.

```
TsgcHTTP_OpenAI_JSON *OpenAI = new TsgcHTTP_OpenAI_JSON(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

TsgcOpenAIClass_Request_ChatCompletion *oRequest = new TsgcOpenAIClass_Request_ChatCompletion();
try
{
  oRequest->Model = "gpt-3.5-turbo";
  TsgcOpenAIClass_Request_Completion_Message *oMessage = new TsgcOpenAIClass_Request_Completion_Message();
  oMessage->Role = "user";
  oMessage->Content = "Hello!";
  oMessages = oRequest->Messages;
  SetLength(oMessages, 1);
  oMessages[0] = oMessage;
  oRequest->Messages = oMessages;
  oRequest->Temperature = 1;
  oRequest->N = 2;
  TsgcOpenAIClass_Response_ChatCompletion *oResponse = OpenAI->CreateChatCompletion(oRequest);

  if (Length(oResponse->Choices) > 0)
  {
    ShowMessage(oResponse->Choices[0]->_Message->Content);
  }
}
__finally
{
  delete oRequest;
  delete oResponse;
}
```

OpenAI | Edit

Given a prompt and an instruction, the model will return an edited version of the prompt.

Simple Example

Tell OpenAI to fix the spelling mistakes of a prompt.

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";
ShowMessage(OpenAI->_CreateEdit("text-davinci-edit-001", "Fix the spelling mistakes", "What day of the wek is it?
```

Advanced Example

Tell OpenAI to fix the spelling mistakes of a prompt, with more random output and generate 2 completions for each prompt.

```
TsgcHTTP_OpenAI_JSON *OpenAI = new TsgcHTTP_OpenAI_JSON(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

TsgcOpenAIClass_Request_Edit *oRequest = new TsgcOpenAIClass_Request_Edit();
try
{
  oRequest->Model = "text-davinci-edit-001";
  oRequest->Input = "What day of the wek is it?";
  oRequest->Instruction = "Fix the spelling mistakes";
  oRequest->Temperature = 1;
  oRequest->N = 2;
  TsgcOpenAIClass_Response_Edit *oResponse = OpenAI->CreateEdit(oRequest);

  if (Length(oResponse->Choices) > 0)
  {
    ShowMessage(oResponse->Choices[0]->Text);
  }
}
__finally
{
  delete oRequest;
  delete oResponse;
}
```

OpenAI | Audio

Create Transcription

Transcribes audio into the input language.

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

ShowMessage(OpenAI->_CreateTranscriptionFromFile("whisper-1", "c:\media\audio.mp3"));
```

Create Translation

Translates audio to English.

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

ShowMessage(OpenAI->_CreateTranslationFromFile("whisper-1", "c:\media\audio.mp3"));
```

OpenAI | Moderation

Given an input text, outputs if the model classifies it as violating OpenAI's content policy.

Simple Example

Moderate the following text

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

ShowMessage(OpenAI->_CreateModeration("I want to kill them."));
```

Advanced Example

Moderate the following text choosing the model.

```
TsgcHTTP_OpenAI_JSON *OpenAI = new TsgcHTTP_OpenAI_JSON(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

TsgcOpenAIClass_Request_Moderation *oRequest = new TsgcOpenAIClass_Request_Moderation();
try
{
  oRequest->Model = "text-moderation-latest";
  oRequest->Input = "I want to kill them.";
  TsgcOpenAIClass_Response_Moderation *oResponse = OpenAI->CreateModeration(oRequest);

  if (Length(oResponse->results) > 0)
  {
    ShowMessage(oResponse->results[0]->flagged);
  }
}
__finally
{
  delete oRequest;
  delete oResponse;
}
```

OpenAI | RealTime

You can use the Realtime API for transcription-only use cases, either with input from a microphone or from a file. For example, you can use it to generate subtitles or transcripts in real-time. With the transcription-only mode, the model will not generate responses.

To use the Realtime API for transcription, you need to create a transcription session, connecting via WebSockets. Use the component **TsgcWSAPI_OpenAI** and **TsgcWebSocketClient** to start a new transcription session.

Find below an example of real-time transcription using OpenAI API.

```
TsgcWebSocketClient *WSClient = new TsgcWebSocketClient(NULL);
TsgcAudioRecorderWave *oAudio = new TsgcAudioRecorderWave(NULL);
TsgcWSAPI_OpenAI *OpenAI = new TsgcWSAPI_OpenAI(NULL);
OpenAI->Client = WSClient;
OpenAI->AudioRecorder = oAudio;
OpenAI->OpenAIOptions->APIKey = "your-api-key-here";
OpenAI->OpenAIOptions->Method = sgcoaimTranscription;
OpenAI->OpenAIOptions->Provider = sgcoaipOpenAI;
OpenAI->InputAudio->Language = "en";
OpenAI->InputAudio->Model = "whisper-1";
void __fastcall TForm1::OnOpenAIAudioTranscriptionCompleted(TObject *Sender, TsgcWSOpenAIConversation_Item_Comple
    Log("#transcription_completed: " + aItem->Transcript);
}
```

Send Audio Manually

The component allows you to send audio files manually. You can use the method AppendInputAudioBuffer and pass the audio as a TStream. The audio format must be 24 kHz mono PCM (only a rate of 24000 is supported).

OpenAI | Responses

The Responses API is the core API for interacting with OpenAI models. It supports text and image inputs, tool use, streaming, and structured outputs.

Simple Example

Create a response using the Responses API.

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";
ShowMessage(OpenAI->_CreateResponse("gpt-4o", "What is the capital of France?"));
```

Methods

- **CreateResponse:** Creates a model response given a model and input.
  - **Model:** ID of the model to use (e.g. gpt-4o, gpt-4o-mini).
  - **Input:** Text or structured input for the model.
- **RetrieveResponse:** Retrieves a previously created response by its ID.
  - **ResponseId:** The ID of the response to retrieve.
- **DeleteResponse:** Deletes a response by its ID.
  - **ResponseId:** The ID of the response to delete.
- **CancelResponse:** Cancels an in-progress response.
  - **ResponseId:** The ID of the response to cancel.
- **ListInputItems:** Returns a list of input items for a given response.
  - **ResponseId:** The ID of the response.

OpenAI | Speech

Generate spoken audio from text using the Text-to-Speech (TTS) API. Supports multiple voices and output formats.

Simple Example

Generate speech from text and save to a file stream.

```
OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

TFileStream *oStream = new TFileStream("output.mp3", fmCreate);
try
{
  OpenAI->_CreateSpeech("tts-1", "Hello, how are you?", "alloy", oStream);
}
__finally
{
  delete oStream;
}
```

Methods

- **CreateSpeech:** Generates audio from the input text.
  - **Model:** ID of the model to use (tts-1 or tts-1-hd).
  - **Input:** The text to generate audio for. Maximum 4096 characters.
  - **Voice:** The voice to use (alloy, echo, fable, onyx, nova, shimmer).
  - **ResponseStream:** The stream where the audio data will be written.
  - **ResponseFormat:** (optional) The audio format: mp3 (default), opus, aac, or flac.
  - **Speed:** (optional) The speed of the generated audio (0.25 to 4.0, default 1.0).

OpenAI | Fine-Tuning

Manage fine-tuning jobs to tailor a model to your specific training data. Fine-tuning lets you get more out of the models by providing higher quality results, the ability to train on more examples, and cost savings.

Simple Example

Create a fine-tuning job and list existing jobs.

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

// Create a fine-tuning job
ShowMessage(OpenAI->_CreateFineTuningJob("gpt-4o-mini-2024-07-18", "file-abc123"));

// List fine-tuning jobs
ShowMessage(OpenAI->_ListFineTuningJobs());
```

Methods

- **CreateFineTuningJob:** Creates a fine-tuning job which begins the process of creating a new model from a given dataset.
  - **Model:** The name of the model to fine-tune.
  - **TrainingFile:** The ID of an uploaded file that contains training data.
- **ListFineTuningJobs:** List your organization's fine-tuning jobs.
- **RetrieveFineTuningJob:** Get info about a fine-tuning job.
  - **JobId:** The ID of the fine-tuning job.
- **CancelFineTuningJob:** Immediately cancel a fine-tuning job.
  - **JobId:** The ID of the fine-tuning job to cancel.
- **ListFineTuningJobEvents:** Get status updates for a fine-tuning job.
  - **JobId:** The ID of the fine-tuning job.
- **ListFineTuningJobCheckpoints:** List checkpoints for a fine-tuning job.
  - **JobId:** The ID of the fine-tuning job.

OpenAI | Batch

Create large batches of API requests to run asynchronously. The Batch API returns completions within 24 hours at a 50% discount.

Simple Example

Create a batch job and retrieve its status.

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

// Create a batch
ShowMessage(OpenAI->_CreateBatch("file-abc123", "/v1/chat/completions"));

// List batches
ShowMessage(OpenAI->_ListBatches());
```

Methods

- **CreateBatch:** Creates and executes a batch from an uploaded file of requests.
  - **InputFileId:** The ID of an uploaded file that contains requests for the batch.
  - **Endpoint:** The endpoint to be used for all requests in the batch (e.g. /v1/chat/completions).
  - **CompletionWindow:** (optional) The time frame within which the batch should be processed. Default is 24h.
- **RetrieveBatch:** Retrieves a batch by its ID.
  - **BatchId:** The ID of the batch to retrieve.
- **ListBatches:** List your organization's batches.
- **CancelBatch:** Cancels an in-progress batch.
  - **BatchId:** The ID of the batch to cancel.

OpenAI | Uploads

Upload large files in parts. The Uploads API allows you to upload files larger than 512 MB by splitting them into parts and uploading each part separately.

Simple Example

Create an upload, add a part, and complete it.

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

// Create an upload
AnsiString vUploadResponse = OpenAI->_CreateUpload("training_data.jsonl",
  "fine-tune", 2147483648, "application/jsonl");

// Add a part
AnsiString vPartResponse = OpenAI->_AddUploadPart("upload_abc123", "part1.jsonl");

// Complete the upload
ShowMessage(OpenAI->_CompleteUpload("upload_abc123", "[\"part_abc123\"]"));
```

Methods

- **CreateUpload:** Creates an intermediate Upload object to begin a multi-part upload.
  - **Filename:** The name of the file to upload.
  - **Purpose:** The intended purpose of the uploaded file (e.g. fine-tune).
  - **Bytes:** The number of bytes in the file.
  - **MimeType:** The MIME type of the file (e.g. application/jsonl).
- **AddUploadPart:** Adds a part to an Upload object.
  - **UploadId:** The ID of the Upload.
  - **Filename:** The chunk of bytes for this part.
- **CompleteUpload:** Completes the Upload. The uploaded file is usable only after completion.
  - **UploadId:** The ID of the Upload.
  - **PartIds:** The ordered list of Part IDs as a JSON array string.
- **CancelUpload:** Cancels the Upload. No parts may be added after cancellation.
  - **UploadId:** The ID of the Upload to cancel.

OpenAI Applications

Overview

Using the OpenAI API you can build a wide range of applications, here are some examples:

- **Chatbots and Virtual Assistants:** Applications that can converse with humans in a natural, human-like manner. These can be used for customer support, handling queries, and providing information on a website or mobile app.
- **Content Generation:** Applications that can generate human-like text such as articles, blog posts, or reports. For instance, GPT-3 can be used to automate content creation for social media, generate code, or create SEO-friendly content.
- **Translation Services:** Applications that can translate text from one language to another.
- **Tutoring and Education:** AI can be used to create personalized learning experiences, help with homework, or explain complex concepts in simple language.
- **Games:** OpenAI can be used to create immersive and interactive games, especially those that involve conversational characters or complex narratives.
- **Sentiment Analysis:** Analyzing and categorizing the sentiments expressed in text data can be useful for market research, brand monitoring, and understanding customer feedback.
- **Personalized Recommendations:** Based on users' past behaviors and preferences, AI can generate personalized recommendations for products, services, or content.
- **Text Completion:** Completing user's sentences or helping with writing assistance in email clients or word processing software.
- **Speech Recognition:** Transcribe spoken language into written text, useful in transcription services, voice assistants, and more.
- **Medical and Legal Advisory:** Although not capable of replacing professional advice, AI models can provide preliminary guidance or suggestions based on given inputs.

Components

Find below a list of the available components and a short description about them.

- **TsgcAIOpenAIAssistant:** An Assistant has instructions and can leverage models, tools, and files to respond to user queries.
- **TsgcAIOpenAIChatBot:** a ChatBot that listens to speech and converts it to text using the OpenAI Whisper API. This text is sent to the ChatCompletion API, which provides a response from OpenAI, and this response is converted from text to speech.
- **TsgcAIOpenAITranslator:** a Translator application that allows you to translate any language speech to English and listen to the translation using any of the TextToSpeech components available.
- **TsgcAIOpenAIEmbeddings:** this component is used to build AI applications with customized data, example: a chatbot with our product data. The following Databases are supported:
    - **TsgcAIDatabaseVectorPinecone:** supports the Pinecone Database Vector.

The following components are used for capturing audio from the microphone, playing audio files, and converting text to speech.

- **TsgcAudioRecorderMCI:** (for Windows only) this component allows you to access the microphone and convert the speech to a wave file.
- **TsgcAudioPlayerMCI:** (for Windows only) this component allows you to play an mp3 file.
- **TsgcTextToSpeechSystem:** (for Windows only) converts text to speech without the need to use an external mp3 file.
- **TsgcTextToSpeechGoogle:** converts text to speech using any of the Google Cloud voices available.
- **TsgcTextToSpeechAmazon:** converts text to speech using any of the Amazon AWS voices available.

TsgcAIOpenAIAssistant

The Assistants API allows you to build AI assistants within your own applications.

Introduction

The Assistants API allows you to build AI assistants within your own applications. An Assistant has instructions and can leverage models, tools, and files to respond to user queries. The Assistants API currently supports three types of tools: Code Interpreter, File Search, and Function calling.

Overview

A typical integration of the Assistants API has the following flow:

- Create an Assistant by defining its custom instructions and picking a model. If helpful, add files and enable tools like Code Interpreter, File Search, and Function calling.
- Create a Thread when a user starts a conversation.
- Add Messages to the Thread as the user asks questions.
- Run the Assistant on the Thread to generate a response by calling the model and the tools.

Step 1: Create an Assistant

An Assistant represents an entity that can be configured to respond to a user's messages using several parameters like model, instructions, and tools.

```
// Create a new assistant
TsgcAIOpenAIAssistant *oAssistant = new TsgcAIOpenAIAssistant(NULL);
// Set your API key
oAssistant->OpenAIOptions->ApiKey = txtAPIKey->Text;
// Assistant options
oAssistant->AssistantOptions->Name = "Math Tutor";
oAssistant->AssistantOptions->Instructions->Text = "You are a personal math tutor. Write and run code to answ
oAssistant->AssistantOptions->Model = "gpt-4o";
// Create the assistant
oAssistant->CreateAssistant();
```

Step 2: Create a Thread

A Thread represents a conversation between a user and one or many Assistants. You can create a Thread when a user (or your AI application) starts a conversation with your Assistant.

```
oThread = oAssistant->CreateThread();
```

Step 3: Add a Message to the Thread and Run

The contents of the messages your users or applications create are added as Message objects to the Thread. Messages can contain both text and files. There is no limit to the number of Messages you can add to Threads — we smartly truncate any context that does not fit into the model's context window.

Once all the user Messages have been added to the Thread, you can Run the Thread with any Assistant. Creating a Run uses the model and tools associated with the Assistant to generate a response. These responses are added to the Thread as assistant Messages.

```
void SendMessage(TsgcAIOpenAIAssistant *oAssistant, TsgcAIClass_Thread *oThread, const UnicodeString aMessage
{
    int i;
    TsgcOpenAIClass_Message *oMessage;
    TsgcOpenAIClass_Response_List_Messages *oMessages;
    TsgcOpenAIClass_Run *oRun;
    DoLog(L"[user]: " + aMessage);
    oMessage = oAssistant->CreateMessageText(oThread->Id, aMessage);
    if (oMessage != NULL)
    {
        oRun = oAssistant->CreateRunAndWait(oThread->Id);
        if (oRun != NULL)
        {
            oMessages = oAssistant->GetMessages(oThread->Id, oRun->Id);
            if (oMessages != NULL && oMessages->Messages.Length > 0)
            {
                for (i = 0; i < oMessages->Messages.Length; i++)
                {
                    DoLog(L"[assistant]: " + DoFormatResponse(oMessages->Messages[i]->ContentText + L"\r\n"))
                }
            }
        }
    }
}
```

Reference

- Properties — 5 properties available on the component.
- Methods — 9 public methods.
- Events — 8 events you can handle.

TsgcAIOpenAIAssistant | File Search

File Search enhances the Assistant's capabilities by integrating knowledge from external sources, such as proprietary product information or documents provided by users. Here's how it works:

- **Document Processing:** OpenAI automatically processes and segments (or "chunks") your documents into manageable pieces of information.
- **Embedding Creation:** These chunks are transformed into embeddings—mathematical representations of the content—that capture their meaning and context.
- **Storage and Search:** The system stores these embeddings, enabling both vector search (to find semantically similar content) and keyword search (to locate exact matches).

This combination ensures that the Assistant can retrieve and leverage relevant content to provide accurate and informed answers to user queries.

Step 1: Create a new Assistant with File Search Enabled

Create a new assistant with file_search enabled in the tools parameter of the Assistant. Once the file_search tool is enabled, the model decides when to retrieve content based on user messages.

```
TsgcAIOpenAIAssistant* Assistant = new TsgcAIOpenAIAssistant(nullptr);
Assistant->OpenAIOptions->ApiKey = "sk-askdjfalskdjfl23kjkjasdefasdfj";
Assistant->AssistantOptions->Name = "sgcWebSockets HelpDesk";
Assistant->AssistantOptions->Instructions->Text =
    "You are a sgcWebSockets HelpDesk Agent. "
    "Answer questions briefly, in a sentence or less. When asked a question, use the manual to answer the questic
Assistant->AssistantOptions->Model = "gpt-4o-mini";
Assistant->AssistantOptions->Tools->FileSearch->Enabled = true;
Assistant->AssistantOptions->Tools->CodeInterpreter->Enabled = false;
```

Step 2: Upload files and add them to a Vector Store

To access your files, the file_search tool uses the Vector Store object. Upload your files and create a Vector Store to contain them.

```
void UploadFile()
{
    TOpenDialog* oDialog = new TOpenDialog(nullptr);
    try
    {
        if (oDialog->Execute())
        {
            Screen->Cursor = crHourGlass; // Change cursor to hourglass
            try
            {
                Assistant->UploadVectorStoreFile("sgcVectorStore", oDialog->FileName.c_str());
            }
            __finally
            {
                Screen->Cursor = crDefault; // Reset cursor to default
            }
        }
    }
    __finally
    {
        delete oDialog; // Clean up dialog
    }
}
```

Step 3: Create a run and check the output

Now, create a Run and observe that the model uses the File Search tool to provide a response to the user's question.

```
void SendMessage()
{
    int i;
    TsgcOpenAIClass_Message* oMessage = nullptr;
    TsgcOpenAIClass_Response_List_Messages* oMessages = nullptr;
    TsgcOpenAIClass_Run* oRun = nullptr;
    DoLog("[user]: " + memoMessage->Lines->Text);
    Screen->Cursor = crHourGlass; // Change cursor to hourglass
    try
    {
        oMessage = Assistant->CreateMessageText("thread_id", "Create a WebSocket Client that connects to eSeGeCe
        if (oMessage != nullptr)
        {
            oRun = Assistant->CreateRunAndWait("thread_id");
            if (oRun != nullptr)
            {
                oMessages = Assistant->GetMessages("thread_id", oRun->Id);
                if (oMessages != nullptr && oMessages->Messages.Length > 0)
                {
                    memoMessage->Lines->Clear();
                    for (i = 0; i < oMessages->Messages.Length; i++)
                    {
                        DoLog("[assistant]: " + DoFormatResponse(oMessages->Messages[i].ContentText + "\r\n"));
                    }
                }
            }
        }
    }
    __finally
    {
        Screen->Cursor = crDefault; // Reset cursor to default
    }
}
```

TsgcAIOpenAIAssistant | Streaming

Instead of waiting for the full response from the assistant, you can stream the response using Server-Sent Events. Just pass the parameter **Stream = True** when using the **CreateRun** function and the response will use streaming.

The following events are used to handle the streaming responses:

- **OnStreamRun:** the event is called when there is an update in the run object.
- **OnStreamMessage:** the event is called when there is an update in the message object: created, in-progess, completed...
- **OnStreamMessageDelta:** Occurs when parts of a Message are being streamed.
- **OnStreamDone:** Occurs when a stream ends.
- **OnStreamError:** Occurs when an error occurs. This can happen due to an internal server error or a timeout.

Step 1: Create an Assistant

An Assistant represents an entity that can be configured to respond to a user's messages using several parameters like model, instructions, and tools.

```
// Create a new assistant
TsgcAIOpenAIAssistant *oAssistant = new TsgcAIOpenAIAssistant(NULL);
// Set your API key
oAssistant->OpenAIOptions->ApiKey = txtAPIKey->Text;
// Assistant options
oAssistant->AssistantOptions->Name = "Math Tutor";
oAssistant->AssistantOptions->Instructions->Text = "You are a personal math tutor. Write and run code to answer m
oAssistant->AssistantOptions->Model = "gpt-4o";
// Create the assistant
oAssistant->CreateAssistant();
```

Step 2: Create a Thread

A Thread represents a conversation between a user and one or many Assistants. You can create a Thread when a user (or your AI application) starts a conversation with your Assistant.

```
oThread = oAssistant->CreateThread();
```

Step 3: Add a Message to the Thread and Run using Streaming

The contents of the messages your users or applications create are added as Message objects to the Thread. Messages can contain both text and files. There is no limit to the number of Messages you can add to Threads — we smartly truncate any context that does not fit into the model's context window.

Once all the user Messages have been added to the Thread, you can Run the Thread with any Assistant. Creating a Run uses the model and tools associated with the Assistant to generate a response. The responses are sent using a Server-Sent stream.

```
void SendMessage(TsgcAIOpenAIAssistant *oAssistant, TsgcAIClass_Thread *oThread, const UnicodeString aMessage)
{
    int i;
```

```
    TsgcOpenAIClass_Message *oMessage;
    TsgcOpenAIClass_Response_List_Messages *oMessages;
    TsgcOpenAIClass_Run *oRun;
    DoLog(L"[user]: " + aMessage);
    oMessage = oAssistant->CreateMessageText(oThread->Id, aMessage);
    if (oMessage != NULL)
    {
        oRun = oAssistant->CreateRun(oThread->Id, true);
    }
}
```

Step 4: Handle the Response

Use the event OnStreamMessageDelta to read the server-sent stream message.

```
void __fastcall OnStreamMessageDelta(TObject *Sender, const TsgcOpenAIClass_MessageDelta &aMessageDelta, const St
{
    for (int i = aMessageDelta.Content.Low(); i <= aMessageDelta.Content.High(); i++)
    {
        String vType = aMessageDelta.Content[i]->_Type;
        if (vType == "text")
        {
            String vResponse = static_cast<TsgcOpenAIClass_MessageDeltaContent_Text*>(aMessageDelta.Content[i])->
        }
    }
}
```

TsgcAIOpenAIAssistant Function Calling

Similar to the Chat Completions API, the Assistants API supports function calling. Function calling allows you to describe functions to the Assistants API and have it intelligently return the functions that need to be called along with their arguments.

In this example, we'll create a weather assistant and define two functions, **get_current_temperature** and **get_rain_probability**, as tools that the Assistant can call. In our example that uses parallel function calling, we will ask the Assistant what the weather in San Francisco is like today and the chances of rain. We also show how to output the Assistant's response with streaming.

Step 1 Define Functions

When creating your assistant, you will first define the functions under the tools param of the assistant.

```
TsgcAIOpenAIAssistant *Assistant = new TsgcAIOpenAIAssistant(nullptr);
Assistant->OpenAIOptions->ApiKey = "sk-askdjfalskdjfl23kjkjasdefasdfj";
Assistant->AssistantOptions->Name = "Delphi Weather Bot";
Assistant->AssistantOptions->Instructions->Text = "You are a weather bot. Use the provided functions to answer qu
Assistant->AssistantOptions->Model = "gpt-4o";
Assistant->AssistantOptions->Tools->Functions->Enabled = false;
Assistant->AssistantOptions->Tools->Functions->Functions->Text =
    "[{\"type\":\"function\",\"function\":{\"name\":\"get_current_temperature\",\"description\":\"Get the current
Assistant->AssistantOptions->Tools->FileSearch->Enabled = false;
Assistant->AssistantOptions->Tools->CodeInterpreter->Enabled = false;
```

Step 2: Create a Thread and add Messages

Create a Thread when a user starts a conversation and add Messages to the Thread as the user asks questions.

```
void SendMessage()
{
    int i;
    TsgcOpenAIClass_Message* oMessage = nullptr;
    TsgcOpenAIClass_Response_List_Messages* oMessages = nullptr;
    TsgcOpenAIClass_Run* oRun = nullptr;
    DoLog("[user]: " + memoMessage->Lines->Text);
    Screen->Cursor = crHourGlass; // Change cursor to hourglass
    try
    {
        oMessage = Assistant->CreateMessageText("thread_id", "What is the weather in San Francisco today and the
        if (oMessage != nullptr)
        {
            oRun = Assistant->CreateRunAndWait("thread_id");
            if (oRun != nullptr)
            {
                oMessages = Assistant->GetMessages("thread_id", oRun->Id);
                if (oMessages != nullptr && oMessages->Messages.Length > 0)
                {
                    memoMessage->Lines->Clear();
                    for (i = 0; i < oMessages->Messages.Length; i++)
                    {
                        DoLog("[assistant]: " + DoFormatResponse(oMessages->Messages[i].ContentText + "\r\n"));
                    }
                }
            }
        }
    }
    __finally
    {
        Screen->Cursor = crDefault; // Reset cursor to default
    }
}
```

Step 3: Handle OnFunctionCall Event

When the component detects that a function parameter value is required, the event **OnFunctionCall** is called. Use the **Request._Function** parameter to know the request details and use the **Response.Output** to send the response.

```
void __fastcall TFRMOpenAIAssistant::AssistantFunctionCall(TObject *Sender,
    const TsgcOpenAIClass_ToolCall &aRequest,
    const TsgcHTTPOpenAI_ToolCall_Response &aResponse)
{
    if (aRequest.Function.Name == "get_current_temperature")
        aResponse.Output = 30;
    else if (aRequest.Function.Name == "get_rain_probability")
        aResponse.Output = 10;
}
```

OpenAI Audio

To use OpenAI APIs with voice commands, the following steps are required:

1. The microphone audio must be captured, so a speech-to-text system is needed to get the text that will be sent to OpenAI.
    1. Capturing the microphone audio is done using the component TsgcAudioRecorderMCI.
    2. Once we've captured the audio, it is sent to the OpenAI Whisper API to convert the audio file to text.
2. Once we get the speech-to-text result, we send the text to OpenAI using the ChatCompletion API.
3. The response from OpenAI must then be converted to speech using one of the following components:
    1. TsgcTextToSpeechSystem: (currently only for Windows) uses the Windows text-to-speech from the operating system.
    2. TsgcTextToSpeechGoogle: sends the response from OpenAI to the Google Cloud Servers and an mp3 file is returned which is played by the TsgcAudioPlayerMCI.
    3. TsgcTextToSpeechAmazon: sends the response from OpenAI to the Amazon AWS Servers and an mp3 file is returned which is played by the TsgcAudioPlayerMCI.

Components

The following components are used for capturing audio from the microphone, playing audio files, and converting text to speech.

- **TsgcAudioRecorderMCI:** (for windows only) this component allows you to access the microphone and convert the speech to a wave file.
- **TsgcAudioPlayerMCI:** (for windows only) this component allows you to play an mp3 file.
- **TsgcTextToSpeechSystem:** (for windows only) converts text to speech without the need of using an external mp3 file.
- **TsgcTextToSpeechGoogle:** converts text to speech using any of the Google Cloud voices available.
- **TsgcTextToSpeechAmazon:** converts text to speech using any of the Amazon AWS voices available.

TsgcAudioRecorderMCI

This component is used to capture the microphone audio and store it in a wave file.

Introduction

This component is used to capture the microphone audio and store it in a wave file. Currently only Windows is supported.

Reference

- Properties — 3 properties available on the component.
- Methods — 2 public methods.
- Events — 0 events you can handle.

TsgcAudioPlayerMCI

This component is used to play the mp3 files received from the text-to-speech providers.

Introduction

This component is used to play the mp3 files received from the text-to-speech providers. Currently only Windows is supported.

Reference

- Properties — 1 properties available on the component.
- Methods — 1 public methods.
- Events — 0 events you can handle.

TsgcTextToSpeechSystem

This is the default Text-To-Speech provided by the Operating System, currently only Windows is supported.

Introduction

This is the default Text-To-Speech provided by the Operating System, currently only Windows is supported.

Reference

- Properties — 1 properties available on the component.
- Methods — 1 public methods.
- Events — 0 events you can handle.

TsgcTextToSpeechGoogle

Text-To-Speech is an API provided by Google Cloud that allows you to convert text to mp3 files.

Introduction

Text-To-Speech is an API provided by Google Cloud that allows you to convert text to mp3 files. It requires a Google Cloud account and setting up the Text-To-Speech account.

Once the Text-To-Speech Account is configured, a JSON settings file must be downloaded and set to the property GoogleOptions.Settings.

Reference

- Properties — 3 properties available on the component.
- Methods — 1 public methods.
- Events — 0 events you can handle.

TsgcTextToSpeechAmazon

Text-To-Speech is an API provided by Amazon AWS that allows you to convert text to mp3 files.

Introduction

Text-To-Speech is an API provided by Amazon AWS that allows you to convert text to mp3 files. It requires an Amazon AWS account and setting up the Polly API.

Reference

- Properties — 3 properties available on the component.
- Methods — 1 public methods.
- Events — 0 events you can handle.

TsgcAIChat

TsgcAIChat is a single component that works with any AI provider.

Introduction

TsgcAIChat is a single component that works with any AI provider. Switch providers by changing one property. The component includes built-in conversation history management, making it easy to build chat applications without manually tracking messages.

Supported Providers

- OpenAI
- Anthropic
- Gemini
- DeepSeek
- Ollama
- Grok
- Mistral

Structured Output

Set **ChatOptions.ResponseFormat** to `rfJSON` or to `rfJSONSchema` and call **ChatJSON** to get the answer as a JSON document, whichever provider is selected. TsgcAIChat Structured Output describes the options, what each provider receives and the one combination that is refused.

Documents and PDF

Send PDF documents and images with a message using **ChatWithDocument**, **ChatWithDocumentBase64**, **ChatWithAttachments** and **ChatJSONWithAttachments**. PDF input is accepted by OpenAI, Anthropic and Gemini; DeepSeek, Grok, Mistral and Ollama accept images only. TsgcAIChat Documents and PDF describes the methods and the attachments list, and TsgcAIDocumentExtractor turns documents into validated structured data.

Reference

- Properties — 5 properties available on the component.
- Methods — 10 public methods.
- Events — 3 events you can handle.

TsgcAIChat Structured Output

How TsgcAIChat asks a provider for JSON, and what each provider receives.

Introduction

Structured output asks the provider for a JSON answer instead of prose, so the answer can be read with a JSON parser and mapped onto your own types. It is configured with three sub-properties of ChatOptions and read back with ChatJSON. Every provider the component supports is covered, each one with its own request shape, which the component builds for you.

ResponseFormat is **rfText** by default, and with that value the request is byte for byte the request the component always sent, so nothing changes for existing code.

Quick Start

Set the format, set the schema and call ChatJSON:

```
oChat->Provider = aicpOpenAI;
oChat->ChatOptions->ApiKey = "API_KEY";
oChat->ChatOptions->Model = "gpt-4o-mini";
oChat->ChatOptions->ResponseFormat = rfJSONSchema;
oChat->ChatOptions->JSONSchema = vSchema;
oChat->ChatOptions->JSONSchemaName = "weather";

String vJSON = oChat->ChatJSON("Weather in Barcelona right now");
```

The Options

Property	Type	Default	Description
ResponseFormat	TsgcAIChatResponseFormat	rfText	rfText, rfJSON or rfJSONSchema.
JSONSchema	string	''	The JSON Schema document. Required when ResponseFormat is rfJSONSchema, and it must itself be valid JSON.
JSONSchemaName	string	'response'	Name sent with the schema. OpenAI, Grok and Mistral require one, and it is also the name of the forced tool of Anthropic. An empty value falls back to response.

ResponseFormat	What is asked for

rfText	The request is built exactly as it was built before structured output existed. This is the default.
rfJSON	Any valid JSON document is asked for. No schema is sent.
rfJSONSchema	A document that validates against **ChatOptions.JSONSchema** is asked for.

What Each Provider Receives

Provider	Request
aicpOpenAI	`response_format` with `type: json_object` for rfJSON, and with `type: json_schema` plus `{ name, schema, strict: true }` for rfJSONSchema.
aicpGrok	The same request as OpenAI.
aicpMistral	The same request as OpenAI.
aicpOllama	The same request as OpenAI.
aicpDeepSeek	DeepSeek implements no `json_schema` mode, so both formats send `response_format` with `type: json_object`. With rfJSONSchema the schema travels in the system message instead. The instruction always contains the word json in lower-case, which DeepSeek requires when json_object is asked for.
aicpAnthropic	Anthropic has no JSON mode. rfJSON appends the instruction to the system message. rfJSONSchema sends one tool whose `input_schema` is the schema, plus `tool_choice` with that tool name, so the model is forced to call it, and the answer is read from the `input` object of the `tool_use` block.
aicpGemini	`generationConfig.responseMimeType` is set to `application/json`, and rfJSON-Schema adds `generationConfig.responseSchema`.

The instruction the two providers without a native schema mode receive is appended to **SystemMessage** for that request only. Your own SystemMessage is kept and the instruction is added on a new line.

Reading the Answer

ChatJSON sends the message with the configured format, reads the answer, checks that it parses as JSON and returns the JSON text. When it does not parse, the question is sent once more with the parse error appended, so the model can correct itself, and a second unparseable answer raises **EsgcAIChat**. Markdown fences are not stripped: an answer wrapped in a code fence is not valid JSON, so it goes through the same retry.

Unlike **Chat**, which reports a transport or provider error through OnChatError and returns an empty string, ChatJ-SON lets those errors propagate to the caller. Wrap the call in a try block:

```
try
{
  String vJSON = oChat->ChatJSON(vQuestion);
}
catch (Exception &E)
{
  // EsgcAIChat: not JSON after one retry; anything else: transport
}
```

An answer that parses is added to the history as the assistant turn and raised through **OnChatMessage**, exactly as Chat does. An answer that does not parse is not added, so the history never keeps a broken document.

Exceptions and Limits

EsgcAIChat is the exception class of the component, declared in **sgcAI_Chat**. It is raised in four cases:

- ResponseFormat is rfJSONSchema and **JSONSchema** is empty.
- JSONSchema is not a valid JSON document. The message carries the parser error.
- ChatJSON could not parse the answer, twice.
- **ChatStream** is called while Provider is aicpAnthropic and ResponseFormat is rfJSONSchema. The forced tool arrives as `input_json_delta` chunks, which the component does not assemble, so the combination is refused instead of returning an empty answer. Use ChatJSON, or set ResponseFormat to rfJSON.

Every other provider streams with structured output. The validation of the schema happens on the first request that uses it, not when the property is assigned.

BaseUrl on Every Provider

ChatOptions.BaseUrl replaces the vendor endpoint base of the selected provider, so a gateway, a proxy or a local OpenAI compatible server can be used with any of them. It used to be honoured by Ollama alone. Trailing slashes are trimmed, and an empty value keeps the vendor endpoint untouched. With aicpOllama the value is the Host of the Ollama options, as before.

```
oChat->Provider = aicpOpenAI;
oChat->ChatOptions->BaseUrl = "https://gateway.local/openai";
oChat->ChatOptions->ApiKey = "API_KEY";
```

See Also

- TsgcAIChat
- ChatOptions
- ChatJSON

TsgcAIChat Documents and PDF

How TsgcAIChat sends PDF documents and images with a message.

Introduction

TsgcAIChat can send PDF documents and images together with a user message, so the model answers about their content: summarise a contract, answer a question about a report or read the total of an invoice. The documents are sent as base64 data with their media type, and they are stored in the conversation history together with the message.

Methods

Method	Description
ChatWithDocument	Sends one PDF or image file with the message.
ChatWithDocument-Base64	Sends one document already encoded as base64, with its media type.
ChatWithAttachments	Sends several documents, in order, with the message.
ChatJSONWithAttach-ments	Same as ChatJSON, with the documents on the user message, to get the answer as JSON.

Supported Providers

Images are accepted by every provider. PDF documents are accepted by OpenAI, Anthropic and Gemini; DeepSeek, Grok, Mistral and Ollama accept images only, and sending them a PDF raises **EsgcAIChat**, which ChatWithDocument and ChatWithAttachments report through **OnChatError**. The model must support vision (for example gpt-4o-mini, a Claude model or a Gemini model).

Provider	PDF	Images
aicpOpenAI	Yes	Yes
aicpAnthropic	Yes	Yes
aicpGemini	Yes	Yes
aicpDeepSeek	No	Yes
aicpGrok	No	Yes
aicpMistral	No	Yes
aicpOllama	No	Yes

TsgcAIChatAttachments

An owning list of **TsgcAIChatAttachment** items, each one with **Base64**, **MediaType** and **FileName** properties. It is declared in **sgcAI_Chat**.

Member	Description
`function Add(const aBase64, aMediaType: string; const aFileName: string = ''): TsgcAIChatAttachment;`	Adds a document already encoded as base64. aMediaType is for example `application/pdf` or `image/png`.
`function AddFile(const aFileName: string): TsgcAIChatAttachment;`	Reads the file, encodes it as base64 and takes the media type from the extension: .pdf, .png, .jpg, .jpeg, .gif or .webp. Any other extension raises **EsgcAIChat**.
`procedure Clear;`	Removes and frees every attachment.
`property Count: Integer;`	Number of attachments.
`property Items[aIndex: Integer]: TsgcAIChatAttachment;`	Attachment at a position. It is the default property.

The list passed to ChatWithAttachments or ChatJSONWithAttachments stays owned by the caller. The attachments are copied into the history, so the list can be freed after the call and the next messages of the conversation can still refer to the document.

Example

```
oChat->Provider = aicpAnthropic;
oChat->ChatOptions->ApiKey = "API_KEY";
oChat->ChatOptions->Model = "claude-sonnet-4-20250514";
ShowMessage(oChat->ChatWithDocument("Summarise this contract", "contract.pdf"));

TsgcAIChatAttachments *oAttachments = new TsgcAIChatAttachments();
try
{
  oAttachments->AddFile("page1.png");
  oAttachments->AddFile("page2.png");
  ShowMessage(oChat->ChatWithAttachments("Compare both pages", oAttachments));
}
__finally
{
  delete oAttachments;
}
```

To turn documents into validated structured data (invoices, receipts, identity documents), use TsgcAIDocumentExtractor, which is built on these methods.

See Also

- TsgcAIChat
- TsgcAIChat Structured Output
- TsgcAIDocumentExtractor

TsgcAIDocumentExtractor

Extracts structured data from PDF documents and images with any AI provider.

Introduction

TsgcAIDocumentExtractor reads PDF documents and images (invoices, receipts, identity documents or any document you describe with a JSON Schema) and returns the data they contain as a JSON document. It sends the document to an AI model through a linked TsgcAIChat, so every provider of TsgcAIChat can be used, and it checks the answer locally before returning it.

How It Works

- The data to extract is described by Schema, or by a built in Preset: `dpInvoice`, `dpReceipt` or `dpIDCard`.
- The model is asked for a structured answer (`rfJSONSchema`) with a `data` member that follows your schema and, optionally, a `confidence` value per field.
- The data is validated against the schema. When it does not validate, the model is asked again with the list of errors (ValidationRetries).
- The result is returned as JSON and is also available in LastResult, with the validation errors and the confidence of every field.

Supported Documents

PDF, PNG, JPEG, GIF and WEBP files. PDF input depends on the provider selected in the linked TsgcAIChat:

Provider	PDF	Images
aicpOpenAI	Yes	Yes
aicpAnthropic	Yes	Yes
aicpGemini	Yes	Yes
aicpDeepSeek	No	Yes
aicpGrok	No	Yes
aicpMistral	No	Yes
aicpOllama	No	Yes

Schema Rules

With OpenAI the schema is sent in strict mode: every object must declare `"additionalProperties": false`, every property must be listed in `required`, and optional values use a nullable type such as `["string","null"]`. The presets already follow these rules. The local validation checks `type`, `properties`, `required`, `items` and `enum`, recursing into nested objects and arrays.

Local OCR

Link a TsgcAIOCRTesseract or a TsgcAIOCRWindows to OCR and set OCRMode to recognise the text of the images locally, either as a fallback when the vision request fails (`domFallback`) or always (`domAlways`), which lets a text only model extract the data. The local OCR accepts images only.
Two local OCR engines are available. TsgcAIOCRTesseract runs the Tesseract command line tool on Windows, Linux and macOS, and needs Tesseract installed. TsgcAIOCRWindows uses the OCR engine built into Windows 10 and later, so nothing has to be installed or deployed; it reads the OCR language packs installed in Windows.

Quick Start

```
oChat->Provider = aicpOpenAI;
oChat->ChatOptions->ApiKey = "API_KEY";
oChat->ChatOptions->Model = "gpt-4o-mini";

oExtractor->AIChat = oChat;
oExtractor->Preset = dpInvoice;
String vJSON = oExtractor->ExtractFile("invoice.pdf");
```

The step by step guide Extract data from an invoice shows a complete example, with confidence checks and the OCR fallback.

Reference

- Properties: 13 properties available on the component.
- Methods: 5 public methods.
- Events: 3 events you can handle.

See Also

- TsgcAIChat
- TsgcAIOCRTesseract
- TsgcAIOCRWindows
- Extract data from an invoice
- TsgcAIChat Documents and PDF

TsgcAIOCRTesseract

Local OCR based on the Tesseract command line tool.

Introduction

TsgcAIOCRTesseract recognises the text of an image with the Tesseract OCR command line tool. It is the local OCR used by TsgcAIDocumentExtractor when **OCRMode** is `domFallback` or `domAlways`, and it can also be used on its own.

Requirements

- Tesseract must be installed on the machine, with the trained data of every language you use. The component runs the `tesseract` executable and reads its standard output; nothing is linked into your application.
- Images only: PNG, JPEG, TIFF, BMP and the other formats your Tesseract build reads. A PDF raises **EsgcAIOCR**; convert its pages to images first.
- Windows, Linux and macOS. Android and iOS are not supported. On macOS the Delphi and C++Builder component does not apply **TimeoutMs**, the call waits until Tesseract ends. The .NET component applies it on every OS.

Properties

Property	Type	Default	Description
ExecutablePath	string	tesseract	Path of the Tesseract executable. The default finds it through the PATH.
Language	string	eng	Tesseract language codes, joined with + for several languages (for example `eng+spa`). Passed as `-l`.
PageSegMode	Integer	3	Tesseract page segmentation mode, passed as `--psm`. 3 is fully automatic, 6 treats the image as one block of text.
TimeoutMs	Integer	60000	Maximum time for one recognition, in milliseconds. 0 waits without limit. Not applied on macOS in Delphi and C++Builder; .NET applies it on every OS.
ExtraArguments	string		Extra command line arguments appended as they are, for example `--oem 1`.

Methods

Method	Description

`function RecognizeFile(const aFileName: string): string;`	Returns the text of the image. Raises **EsgcAIOCR** when the file does not exist, when it is a PDF, or when Tesseract ends with an error or does not end within TimeoutMs.
`function RecognizeStream(const aStream: TStream; const aExtension: string): string;`	Writes the stream to a temporary file with the given extension, recognises it and deletes the file.
`function IsAvailable: Boolean;`	Runs `tesseract --version` and returns True when it succeeds. Use it to check the installation before the first extraction.

Events

Event	Description
`OnOCRText(Sender: TObject; const aFileName: string; var aText: string)`	Raised with the recognised text before it is returned, so it can be cleaned or corrected.

Example

```
TsgcAIOCRTesseract *oOCR = new TsgcAIOCRTesseract(NULL);
oOCR->Language = "eng+spa";
if (!oOCR->IsAvailable())
  throw Exception("Tesseract is not installed");
ShowMessage(oOCR->RecognizeFile("receipt.png"));

oExtractor->OCR = oOCR;
oExtractor->OCRMode = domFallback;
```

TsgcAIOCRTesseract is available in Delphi, C++Builder and .NET. In .NET the methods are `RecognizeFile(string aFileName)`, `RecognizeStream(Stream aStream, string aExtension)` and `IsAvailable()`, and the `OnOCRText` event passes the text as `ref string aText`.

See Also

- TsgcAIDocumentExtractor
- TsgcAIOCRWindows
- OCRMode

TsgcAIOCRWindows

Local OCR based on the OCR engine built into Windows.

Introduction

TsgcAIOCRWindows recognises the text of an image with the OCR engine included in Windows 10 and later (Windows.Media.Ocr). Nothing has to be installed or deployed with your application: the component uses the OCR language packs already installed in Windows. It can be linked to TsgcAIDocumentExtractor as the local OCR used when **OCRMode** is `domFallback` or `domAlways`, in place of TsgcAIOCRTesseract, and it can also be used on its own.

Requirements

- Windows 10 or later. On older Windows versions **IsAvailable** returns False and **RecognizeFile** raises **EsgcAIOCR**. On other operating systems the component raises **EsgcAIOCR**.
- The OCR language pack of every language you use. Windows installs it with the language (Settings, Time and language, Language and region), or it can be added on its own, for example with `Add-WindowsCapability -Online -Name "Language.OCR~~~en-US~0.0.1.0"` from an elevated PowerShell. **GetAvailableLanguages** lists the languages installed.
- Images only: PNG, JPEG, BMP, GIF and TIFF. A PDF or any other file type raises **EsgcAIOCR**; convert the PDF pages to images first.

Properties

Property	Type	Default	Description
Language	string		BCP-47 language tag, for example `en-US` or `es-ES`. Empty uses the languages of the user profile. Tesseract codes such as `eng` or `eng+spa` are rejected with **EsgcAIOCR**, and so is a language whose OCR pack is not installed.
TimeoutMs	Integer	60000	Maximum time to decode the image and recognise the text, in milliseconds. 0 waits without limit.

Methods

Method	Description
`function RecognizeFile(const aFileName: string): string;`	Returns the text of the image, one line of text per recognised line. Raises **EsgcAIOCR** when the file does not exist, when it is not a supported image, when the language is not valid or not installed, or when the recognition does not end within TimeoutMs.

`function RecognizeStream(const aStream: TStream; const aExtension: string): string;`	Writes the stream to a temporary file with the given extension, recognises it and deletes the file.
`function IsAvailable: Boolean;`	Returns True when the Windows OCR engine can be created on this machine, which needs Windows 10 or later and at least one OCR language installed. Use it to check the machine before the first extraction.
`procedure GetAvailableLanguages(const aList: TStrings);`	Fills the list with the BCP-47 tags of the OCR languages installed in Windows.

Events

Event	Description
`OnOCRText(Sender: TObject; const aFileName: string; var aText: string)`	Raised with the recognised text before it is returned, so it can be cleaned or corrected.

Example

```
TsgcAIOCRWindows *oOCR = new TsgcAIOCRWindows(NULL);
oOCR->Language = "en-US";
if (!oOCR->IsAvailable())
  throw Exception("Windows OCR is not available");
TStringList *oLanguages = new TStringList();
try
{
  oOCR->GetAvailableLanguages(oLanguages);
  ShowMessage(oLanguages->Text);
}
__finally
{
  delete oLanguages;
}
ShowMessage(oOCR->RecognizeFile("receipt.png"));

oExtractor->OCR = oOCR;
oExtractor->OCRMode = domFallback;
```

TsgcAIOCRWindows is available in Delphi, C++Builder and .NET. In .NET the methods are `RecognizeFile(string aFileName)`, `RecognizeStream(Stream aStream, string aExtension)`, `IsAvailable()` and `GetAvailableLanguages(IList<string> aList)`, and the `OnOCRText` event passes the text as `ref string aText`.

See Also

- TsgcAIDocumentExtractor
- TsgcAIOCRTesseract
- OCRMode

Extract data from an invoice

Step by step guide to read an invoice into JSON with TsgcAIDocumentExtractor.

Introduction

This guide reads a PDF invoice with TsgcAIDocumentExtractor and the built in invoice preset, checks the confidence of the extracted fields and falls back to a local OCR when the model can not read the file.

1. Configure the chat and the extractor

The extractor sends the document through a TsgcAIChat. Pick a provider that accepts PDF documents (OpenAI, Anthropic or Gemini) and a model with vision support. Select `dpInvoice` as preset: no schema needs to be written.

```
#include "sgcAI.hpp"
#include "sgcAI_Chat.hpp"
#include "sgcAI_DocumentExtractor.hpp"

void ExtractInvoice(const String aFileName)
{
  TsgcAIChat *oChat = new TsgcAIChat(NULL);
  TsgcAIDocumentExtractor *oExtractor = new TsgcAIDocumentExtractor(NULL);
  try
  {
    oChat->Provider = aicpOpenAI;
    oChat->ChatOptions->ApiKey = "API_KEY";
    oChat->ChatOptions->Model = "gpt-4o-mini";

    oExtractor->AIChat = oChat;
    oExtractor->Preset = dpInvoice;
    oExtractor->MinConfidence = 0.8;

    String vJSON = oExtractor->ExtractFile(aFileName);
    ShowMessage(vJSON);
    ShowMessage("Lowest confidence: " +
      FloatToStr(oExtractor->LastResult->Confidence));
  }
  __finally
  {
    delete oExtractor;
    delete oChat;
  }
}
```

2. Read the result

ExtractFile returns the `data` object as JSON, the same text as **LastResult.Data**. With the invoice preset it has these members, and every value the invoice does not show is null:

- invoice_number, issue_date, due_date (YYYY-MM-DD) and currency (ISO 4217 code).
- supplier and customer, each one with name, tax_id and address.
- lines: description, quantity, unit_price, discount, tax_rate and amount, in the order of the document.
- taxes: rate, base and amount per tax rate. Rates are percentage numbers (21 for 21%).
- subtotal, tax_total, total, payment_terms and iban.

Parse the text with any JSON library. A shortened answer looks like this:

```
{"invoice_number":"F-2026-0142","issue_date":"2026-09-01","due_date":"2026-10-01",
 "currency":"EUR","supplier":{"name":"Acme SL","tax_id":"B12345678","address":null},
 ...,"subtotal":100,"tax_total":21,"total":121,"payment_terms":null,"iban":null}
```

3. Check the confidence

The model returns a confidence between 0 and 1 for every field it extracted. **LastResult.Confidence** is the lowest one, **LastResult.FieldConfidence** returns the value of one path, and **OnLowConfidence** is raised for every field under **MinConfidence**. Use them to send doubtful invoices to a manual review. The value is the self assessment of the model, not a guarantee.

```
void __fastcall TForm1::oExtractorLowConfidence(TObject *Sender,
  const String aField, double aConfidence)
{
  Memo1->Lines->Add("Review " + aField + " (" + FloatToStr(aConfidence) + ")");
}

// after ExtractFile
if (oExtractor->LastResult->FieldConfidence("total") < 0.9)
  Memo1->Lines->Add("Check the total by hand");
```

4. Handle invalid data

The data is validated against the invoice schema. When it does not validate, the model is asked once more with the list of errors (**ValidationRetries**). If it is still invalid, **EsgcAIDocumentExtractor** is raised with the errors in its message. Set **RaiseOnInvalid** to False to receive the data anyway and read **LastResult.Valid** and **LastResult.ValidationErrors** instead. Transport and provider errors of the chat are raised as they are.

```
try
{
  String vJSON = oExtractor->ExtractFile("invoice.pdf");
}
catch (EsgcAIDocumentExtractor &E)
{
  ShowMessage("Not extracted: " + E.Message);
}
```

5. Scanned invoices and the local OCR

A scanned invoice is an image, which every provider accepts. When the model can not read it, for example because the selected model has no vision support, link a TsgcAIOCRTesseract or a TsgcAIOCRWindows and set **OCRMode** to `domFallback`: the failed request raises **OnOCRFallback** and the data is extracted from the text recognised locally. `domAlways` skips the vision request, so a text only model can be used. The local OCR accepts images only, and it is available in Delphi, C++Builder and .NET. TsgcAIOCRTesseract needs Tesseract installed on the machine.

```
TsgcAIOCRTesseract *oOCR = new TsgcAIOCRTesseract(NULL);
oOCR->Language = "eng";
oExtractor->OCR = oOCR;
oExtractor->OCRMode = domFallback;

String vJSON = oExtractor->ExtractFile("scanned_invoice.png");
if (oExtractor->LastResult->UsedOCR)
  ShowMessage("Extracted from the OCR text");
```

Or use the OCR built into Windows 10 and later, with nothing to install. Its **Language** is a BCP-47 tag such as `en-US`, or empty for the languages of the user profile, and the OCR language pack must be installed in Windows.

```
TsgcAIOCRWindows *oWinOCR = new TsgcAIOCRWindows(NULL);
oWinOCR->Language = "en-US";
oExtractor->OCR = oWinOCR;
oExtractor->OCRMode = domFallback;
```

Your own fields

To add a field the preset does not have, copy the preset schema with **GetPresetSchema**, add the field, and set **Preset** to `dpCustom`. With OpenAI keep the strict rules: list the new field in `required` and give it a nullable type.

See Also

- TsgcAIDocumentExtractor
- Preset
- LastResult
- TsgcAIOCRTesseract
- TsgcAIOCRWindows

TsgcAIVoiceAgent

Realtime voice agent: listens, detects turns, talks back and calls tools, with OpenAI Realtime or Gemini Live.

Introduction

TsgcAIVoiceAgent holds a spoken conversation between a user and an AI model. It captures the microphone, streams the audio to the provider, detects when the user has finished talking, plays the spoken answer, stops talking when the user interrupts, transcribes both sides and runs the tools the model asks for. One component covers both providers, **OpenAI Realtime** (GA API) and **Gemini Live**, and two transports, WebSocket and WebRTC.

How It Works

- Start connects to the Provider with the selected Transport, sends the Instructions, voice, turn detection and tools, and opens the microphone and the speaker.
- The user audio is streamed to the provider. The end of the user turn is detected by the provider (server or semantic VAD), locally (client VAD) or by the application (manual), see TurnDetection.
- The answer is played on the speaker and delivered through OnAudioOutput; the words of both sides arrive in OnTranscript.
- When the user talks over the agent, the answer is stopped at once (BargeIn).
- Tool calls are answered in OnToolCall or automatically by a linked MCP client.

Providers and Transports

Provider	WebSocket	WebRTC	Wire audio
vapOpenAI (OpenAI Realtime)	Yes	Yes (P2P/WebRTC edition)	PCM16 24 kHz, or G.711 8 kHz directly
vapGemini (Gemini Live)	Yes	No	PCM16 16 kHz in, 24 kHz out

The application always works in the format of Audio.Format (PCM16 at 8, 16, 24 or 48 kHz, or G.711 mu-law or A-law at 8 kHz) and the component converts to the format of the provider.

Supported Platforms

Platform	Microphone and speaker	Notes
Windows	Yes (waveIn / waveOut)	
Android	Yes	The application must request the RECORD_AUDIO permission at runtime before Start.
iOS	Yes	Add NSMicrophoneUsageDescription to the Info.plist of the application.

Linux, macOS	No	Set Audio.UseDevice to False and exchange the audio with SendAudio and OnAudioOutput.

Quick Start

```
#include "sgcAI_VoiceAgent.hpp"
#include "sgcAI_VoiceAgent_Types.hpp"

oVoice = new TsgcAIVoiceAgent(NULL);
oVoice->Provider = vapOpenAI;
oVoice->OpenAIOptions->ApiKey = "API_KEY";
oVoice->OpenAIOptions->Voice = "alloy";
oVoice->Instructions = "You are a friendly assistant. Answer briefly.";
oVoice->OnTranscript = oVoiceTranscript;
oVoice->OnError = oVoiceError;
oVoice->Start();
```

```
void __fastcall TForm1::oVoiceTranscript(TObject *Sender,
  TsgcAIVoiceAgentRole aRole, const String aText, bool aFinal)
{
  if (aFinal)
  {
    if (aRole == varUser)
      Memo1->Lines->Add("You: " + aText);
    else
      Memo1->Lines->Add("Agent: " + aText);
  }
}
```

Events

- OnConnect: Fired when the connection with the provider is open.
- OnDisconnect: Fired when the session is closed.
- OnError: Fired when the provider, the transport, the audio device or a tool reports an error.
- OnSessionCreated: Fired when the provider has created the session.
- OnTurnStarted: Fired when the user starts talking or the agent starts answering.
- OnTurnEnded: Fired when the user stops talking or the agent finishes its answer.
- OnBargeIn: Fired when the user interrupts the agent.
- OnTranscript: Fired with the text of what the user and the agent say.
- OnAudioOutput: Fired with every chunk of audio of the agent.
- OnToolCall: Fired when the model calls a tool.
- OnRawMessage: Fired with every JSON message received from the provider.

Guides

- Voice assistant with tools: OnToolCall and MCP tools.
- Barge-in and turn detection: VAD modes, push to talk and interruptions.
- Telephony with G.711: connect the agent to a phone call.

Reference

- Properties: 14 properties available on the component.
- Methods: 7 public methods.
- Events: 11 events you can handle.

See Also

- TsgcAIChat
- TsgcWSAPIClient_MCP
- Voice assistant with tools
- Barge-in and turn detection
- Telephony with G.711

Voice assistant with tools

Let the voice agent call your own functions and the tools of an MCP server.

Introduction

A voice assistant becomes useful when it can act: look up an order, book an appointment, read a sensor. This guide gives TsgcAIVoiceAgent a local tool answered in OnToolCall, and then the tools of an MCP server with no code at all.

1. Declare the tool

Add an item to Tools with the name the model will call, a description that tells the model when to call it, and the JSON Schema of the arguments. Mention the tool in the Instructions too.

```
void __fastcall TForm1::StartWeatherAssistant()
{
  oVoice->Provider = vapOpenAI;
  oVoice->OpenAIOptions->ApiKey = "API_KEY";
  oVoice->Instructions = "You are a weather assistant. "
    "Call get_weather to know the weather of a city before answering.";

  oVoice->Tools->Clear();
  TsgcAIVoiceAgentTool *oTool = oVoice->Tools->Add();
  oTool->Name = "get_weather";
  oTool->Description = "Returns the current weather of a city.";
  oTool->Parameters = "{\"type\":\"object\",\"properties\":"
    "{\"city\":{\"type\":\"string\",\"description\":\"Name of the city\"}},"
    "\"required\":[\"city\"]}";

  oVoice->OnToolCall = oVoiceWeatherToolCall;
  oVoice->OnTranscript = oVoiceTranscript;
  oVoice->Start();
}
```

2. Answer the call

OnToolCall receives the name and the JSON arguments. Return the result as JSON text in `aResult` and set `aHandled` to True. The component sends the result back and asks the model to go on talking, so the user hears the answer a moment later. Keep the handler fast: the agent is silent while it runs.

```
void __fastcall TForm1::oVoiceWeatherToolCall(TObject *Sender,
  const String aCallId, const String aName, const String aArguments,
  String &aResult, bool &aHandled)
{
  if (aName != "get_weather")
    return; // ... not mine: the MCP client, if any, gets the call

  String vCity;
  TsgcJSON *oJSON = new TsgcJSON(NULL);
  try
  {
    oJSON->Read(aArguments);
    if (oJSON->Node["city"] != NULL)
      vCity = oJSON->Node["city"]->Value;
  }
  __finally
  {
    delete oJSON;
  }

  // ... replace with a call to your weather service
```

```
  aResult = "{\"city\":\"" + vCity + "\",\"temperature\":21,\"sky\":\"clear\"}";
  aHandled = true;
}
```

3. Add the tools of an MCP server

Link a TsgcWSAPIClient_MCP to MCPClient. When the session starts the component lists the tools of the server and offers them to the model together with the local ones. Every call that OnToolCall leaves unhandled is executed on the MCP server and its result goes back to the model.

```
TsgcWSAPIClient_MCP *oMCP = new TsgcWSAPIClient_MCP(NULL);
oMCP->MCPOptions->HttpOptions->URL = "https://localhost:5001/mcp";

oVoice->MCPClient = oMCP;
oVoice->Start();
```

Rules

- A local tool wins when a tool of the MCP server has the same name.
- A call that neither OnToolCall nor the MCP client handles is answered with `{"error":"tool not available"}`, so the model can tell the user.
- An exception raised in OnToolCall is reported in OnError and the call is passed on as not handled.
- The tools are sent when the session starts: change Tools or MCPClient before Start.

See Also

- TsgcAIVoiceAgent
- OnToolCall
- Tools
- MCPClient
- TsgcWSAPIClient_MCP

Barge-in and turn detection

Choose how the end of the user turn is detected and how the agent reacts when it is interrupted.

Introduction

A natural conversation needs two things: the agent must know when the user has finished talking, and it must stop talking when the user starts. TurnDetection controls the first and BargeIn the second.

Server VAD

The default mode, `vtdServerVAD`: the provider detects the voice and ends the turn after `SilenceDurationMs` of silence. Raise `Threshold` in noisy places and raise SilenceDurationMs for users who pause while they think.

```
oVoice->TurnDetection->Mode = vtdServerVAD;
oVoice->TurnDetection->Threshold = 0.6;
oVoice->TurnDetection->PrefixPaddingMs = 300;
oVoice->TurnDetection->SilenceDurationMs = 700;
```

Semantic VAD

`vtdSemanticVAD` lets the model decide from the words whether the user has finished, so a pause in the middle of a sentence does not end the turn. `Eagerness` sets how quickly it answers: `low` waits longer, `high` answers sooner.

```
oVoice->TurnDetection->Mode = vtdSemanticVAD;
oVoice->TurnDetection->Eagerness = "low";
```

Client VAD

`vtdClientVAD` detects the speech locally from the energy of the audio. When the user stops, the component commits the audio and asks for the answer itself. Use it when the provider VAD must be off, or to react to speech without the network delay. The level of each 10 ms frame is scaled so that 1 is -20 dBFS.

```
oVoice->TurnDetection->Mode = vtdClientVAD;
oVoice->TurnDetection->Threshold = 0.3;
oVoice->TurnDetection->SilenceDurationMs = 600;
```

Push to talk

With `vtdManual` nothing is detected: the audio sent since the last commit becomes the user turn when the application calls CommitAudio, and CreateResponse asks for the answer.

```
void __fastcall TForm1::btnTalkMouseUp(TObject *Sender)
{
  // ... TurnDetection->Mode is vtdManual
  oVoice->CommitAudio();
```

```
  oVoice->CreateResponse();
}
```

Barge-in

With BargeIn True (the default), when the user starts talking while the agent speaks, the component clears the audio still queued on the speaker, cancels the answer, and truncates the agent message at the point the user really heard, so the model knows what was not said. Then OnBargeIn and OnTurnEnded (varAgent, aInterrupted = True) are raised. A Stop button can do the same with Interrupt.

```
oVoice->BargeIn = true;
oVoice->OnTurnStarted = oVoiceTurnStarted;
oVoice->OnTurnEnded = oVoiceTurnEnded;
oVoice->OnBargeIn = oVoiceBargeIn;
```

```
void __fastcall TForm1::oVoiceTurnEnded(TObject *Sender,
  TsgcAIVoiceAgentRole aRole, bool aInterrupted)
{
  if ((aRole == varAgent) && aInterrupted)
    Memo1->Lines->Add("The agent was interrupted.");
}
```

Echo

On a laptop without headphones the microphone can hear the agent and interrupt it. Use headphones, raise Threshold, or set `TurnDetection.InterruptResponse` and BargeIn to False so the agent always finishes its answer.

See Also

- TsgcAIVoiceAgent
- TurnDetection
- BargeIn
- Interrupt
- OnTurnStarted
- OnTurnEnded

Telephony with G.711

Connect the voice agent to a phone call with G.711 mu-law or A-law audio at 8 kHz.

Introduction

Phone networks and SIP trunks carry G.711 audio at 8 kHz: mu-law (PCMU) in North America and Japan, A-law (PCMA) in the rest of the world. TsgcAIVoiceAgent accepts and produces this audio directly, so the RTP payload of a call can be passed to the agent and back without conversion code.

1. Configure the agent

Set `Audio.Format` to `vafG711ULaw` or `vafG711ALaw` and `Audio.UseDevice` to False, because the audio comes from the call and not from a microphone. With OpenAI the G.711 audio travels to the model as it is (audio/pcmu or audio/pcma); with Gemini the component converts it to PCM16 and back.

```
oVoice->Provider = vapOpenAI;
oVoice->OpenAIOptions->ApiKey = "API_KEY";
oVoice->Instructions = "You answer the phone of a restaurant and take bookings.";
oVoice->Audio->UseDevice = false;
oVoice->Audio->Format = vafG711ULaw;
oVoice->OnAudioOutput = oVoicePhoneAudioOutput;
oVoice->Start();
```

2. Send the caller audio

Pass every audio packet of the caller to SendAudio as it arrives, 20 ms (160 bytes) per RTP packet is fine.

```
void __fastcall TForm1::DoPhoneAudioReceived(const TBytes aData)
{
  // ... G.711 mu-law, 8 kHz, as it arrives from the RTP stream of the call
  oVoice->SendAudio(aData);
}
```

3. Send the agent audio to the caller

OnAudioOutput delivers the answer in the same G.711 format. Packetize it in 20 ms frames and send it on the RTP stream of the call.

```
void __fastcall TForm1::oVoicePhoneAudioOutput(TObject *Sender,
  const TBytes aData)
{
  // ... already G.711 mu-law at 8 kHz: send it back to the caller
  SendToPhone(aData);
}
```

4. End of the call

Stop the agent when the caller hangs up.

```
void __fastcall TForm1::DoPhoneCallEnded()
{
  oVoice->Stop();
}
```

Barge-in on the phone

Without a local speaker the component can not know how much of the answer was played, so it estimates it from the audio delivered through OnAudioOutput. When the caller interrupts, clear the audio you have queued for the call in OnBargeIn, so the caller does not hear the rest of the old answer.

See Also

- TsgcAIVoiceAgent
- Audio
- SendAudio
- OnAudioOutput
- OnBargeIn

TsgcAIOpenAIChatBot

The microphone audio must be captured, so a speech-to-text system is needed to get the text that will be sent to OpenAI.

Introduction

To build a ChatBot with voice commands, the following steps are required:

1. The microphone audio must be captured, so a speech-to-text system is needed to get the text that will be sent to OpenAI.
    1. Capturing the microphone audio is done using the component TsgcAudioRecorderMCI.
    2. Once we've captured the audio, it is sent to the OpenAI Whisper API to convert the audio file to text.
2. Once we get the speech-to-text result, we send the text to OpenAI using the ChatCompletion API.
3. The response from OpenAI must then be converted to speech using one of the following components:
    1. TsgcTextToSpeechSystem: (currently only for Windows) uses the Windows text-to-speech from the operating system.
    2. TsgcTextToSpeechGoogle: sends the response from OpenAI to the Google Cloud Servers and an mp3 file is returned which is played by the TsgcAudioPlayerMCI.
    3. TsgcTextToSpeechAmazon: sends the response from OpenAI to the Amazon AWS Servers and an mp3 file is returned which is played by the TsgcAudioPlayerMCI.

Reference

- Properties — 6 properties available on the component.
- Methods — 4 public methods.
- Events — 4 events you can handle.

TsgcAIOpenAITranslator

The microphone audio must be captured, so a speech-to-text system is needed to get the text that will be sent to OpenAI.

Introduction

To build a Translator with voice commands, the following steps are required:

1. The microphone audio must be captured, so a speech-to-text system is needed to get the text that will be sent to OpenAI.
    1. Capturing the microphone audio is done using the component TsgcAudioRecorderMCI.
    2. Once we've captured the audio, it is sent to the OpenAI Whisper API to convert the audio file to text.
2. Once we get the speech-to-text result, we send the text to OpenAI using the ChatCompletion API.
3. The response from OpenAI must then be converted to speech using one of the following components:
    1. TsgcTextToSpeechSystem: (currently only for Windows) uses the Windows text-to-speech from the operating system.
    2. TsgcTextToSpeechGoogle: sends the response from OpenAI to the Google Cloud Servers and an mp3 file is returned which is played by the TsgcAudioPlayerMCI.
    3. TsgcTextToSpeechAmazon: sends the response from OpenAI to the Amazon AWS Servers and an mp3 file is returned which is played by the TsgcAudioPlayerMCI.

Reference

- Properties — 5 properties available on the component.
- Methods — 2 public methods.
- Events — 3 events you can handle.

TsgcAIOpenAIEmbeddings

Embeddings are a way to represent words, phrases, or even other types of data, like images or audio, in a numerical form.

Introduction

Embeddings are a way to represent words, phrases, or even other types of data, like images or audio, in a numerical form. It's like turning words or data into numbers so that computers can understand and work with them better.

Imagine you have a bunch of words, like "dog," "cat," and "bird." These words have meaning, right? Well, embeddings assign each word a unique set of numbers (vectors) that capture their meaning and relationships to other words.

For example, the word "dog" might be represented as [0.5, 0.2, -0.7], "cat" as [0.8, -0.3, 0.1], and "bird" as [0.3, 0.9, 0.4]. The numbers in the vectors carry information about the characteristics of each word, like whether they are related to animals or how similar they are to each other.

The amazing thing is that embeddings can be learned from large amounts of data, so they can figure out similarities and differences between words automatically. These numerical representations help AI algorithms understand language and make sense of complex patterns, which is crucial in various applications like language translation, sentiment analysis, and recommendation systems. They also make it easier and faster for AI models to process information and provide more accurate results!

Reference

- Properties — 4 properties available on the component.
- Methods — 2 public methods.
- Events — 5 events you can handle.

TsgcAIDatabaseVectorFile

The component stores the database vectors and prompts in two text files.

Introduction

The component stores the database vectors and prompts in two text files. This component should be used only for testing purposes, not for production, because it is not optimized when the number of vectors is high.

Configuration

- **VectorFileOptions:**
  - **InputFilename:** the name of the file used to store the input data.
  - **VectorFilename:** the name of the file used to store the vectors.

Reference

- Properties — 2 properties available on the component.
- Methods — 4 public methods.
- Events — 0 events you can handle.

TsgcAIDatabaseVectorPinecone

TsgcAIDatabaseVectorPinecone component.

Introduction

The component is based on the REST Pinecone API client which allows you to create / update / delete indexes and vectors.

Configuration

- **PineconeOptions:**
  - **ApiKey:** configure here the API Key provided by pinecone which can be obtained from your pinecone account.
  - **Environment:** by default is the free account "us-west4-gcp-free".
  - **LogOptions:** configure here if you want to store the HTTP requests in a text file.

- **PineconeIndexOptions:**
  - **IndexName:** the name of the index used to store or query the data.
  - **ProjectId:** the id of the project.

Reference

- Properties — 3 properties available on the component.
- Methods — 4 public methods.
- Events — 0 events you can handle.

Embeddings | Create Vectors

To use the embeddings, first we must convert our data to vectors.

Example

If you have a PDF file, first convert the PDF file to text and then use the method **CreateEmbeddingsFromFile** to get the vector data. Due to the OpenAI Embeddings size limitation, if the file is too big, the data will be split automatically into chunks, so from one file you can get one or multiple vectors.

Find below a code sample.

```
void ConvertFileToVector()
{
    TOpenDialog* oDialog = new TOpenDialog(NULL);
    try
    {
        oDialog->Filter = "TXT Files|*.txt";
        if (oDialog->Execute())
        {
            TsgcAIOpenAIEmbeddings* oEmbeddings = new TsgcAIOpenAIEmbeddings(NULL);
            try
            {
                TsgcAIDatabaseVectorFile* oFile = new TsgcAIDatabaseVectorFile(NULL);
                try
                {
                    oEmbeddings->Database = oFile;
                    oEmbeddings->OpenAIOptions->ApiKey = "<your api key>";
                    oEmbeddings->CreateEmbeddingsFromFile(oDialog->FileName);
                }
                __finally
                {
                    delete oFile;
                }
            }
            __finally
            {
                delete oEmbeddings;
            }
        }
    }
    __finally
    {
        delete oDialog;
    }
}
```

Embeddings | ChatBot

Once we've converted all our data to vectors, we can start to build our own model. The idea behind it is very simple: every time we ask the bot, first we convert the question to a vector, then we search our database for which vector is most similar to the question, and finally we use the most similar data and add it as context.

```
void AskToChatGPT(const std::string& aQuestion)
{
    TsgcAIOpenAIChatBot* oChatBot = new TsgcAIOpenAIChatBot(NULL);
    try
    {
        oChatBot->OpenAIOptions->ApiKey = "<your api key>";
        TsgcAIOpenAIEmbeddings* oEmbeddings = new TsgcAIOpenAIEmbeddings(NULL);
        try
        {
            oChatBot->Embeddings = oEmbeddings;
            TsgcAIDatabaseVectorFile* oFile = new TsgcAIDatabaseVectorFile(NULL);
            try
            {
                oEmbeddings->Database = oFile;
                std::string vContext = oChatBot->GetEmbedding(aQuestion);
                std::string message = "Answer the question based on the context below.\n\nContext:\n" +
                                      vContext + "\nQuestion:" + aQuestion + "\nAnswer:";
                oChatBot->ChatAsUser(message.c_str());
            }
            __finally
            {
                delete oFile;
            }
        }
        __finally
        {
            delete oEmbeddings;
        }
    }
    __finally
    {
        delete oChatBot;
    }
}
```

Pinecone

Pinecone.io

Pinecone is a vector database that allows you to upload, query, and delete vector data in an easy and powerful way.

Pinecone has a public API that allows third parties to integrate Pinecone into their own applications. The component TsgcHTTP_API_Pinecone is a wrapper over the Pinecone API.

Configuration

Before starting, you must register on the Pinecone website and request an API key. This API key is used to send the API requests and must be set in the property **PineconeOptions.ApiKey** of the TsgcHTTP_API_Pinecone component.

Index Operations

The following methods are supported:

Method	Parameters	Description
IndexesList		This operation returns a list of your Pinecone indexes.
IndexCreate	TsgcHTTP-PineconeIn-dexCreate	This operation creates a Pinecone index. You can use it to specify the measure of similarity, the dimension of vectors to be stored in the index, the numbers of replicas to use, and more.
IndexDescribe	Index Name	Get a description of an index.
IndexDelete	Index Name	This operation deletes an existing index.
IndexConfigure	Index Name, Replicas, PodType	This operation specifies the pod type and number of replicas for an index.

Collection Operations

The following methods are supported:

Method	Parameters	Description
CollectionsList		This operation returns a list of your Pinecone collections.
CollectionCreate	Collection Name, Source	This operation creates a Pinecone collection.
CollectionDescribe	Collection Name	Get a description of a collection.
CollectionDelete	Collection Name	This operation deletes an existing collection.

Vector Operations

The following methods are supported:

Method	Parameters	Description
VectorsDescribeIndexStats	Index Name, Project Id, Filter	The DescribeIndexStats operation returns statistics about the index's contents, including the vector count per namespace and the number of dimensions.
VectorsQuery	Index Name, Project Id, Params	The Query operation searches a namespace, using a query vector. It retrieves the ids of the most similar items in a namespace, along with their similarity scores.
VectorsDelete	Index Name, Project Id, Params	The Delete operation deletes vectors, by id, from a single namespace. You can delete items by their id, from a single namespace.
VectorsFetch	Index Name, Project Id, Ids	The Fetch operation looks up and returns vectors, by ID, from a single namespace. The returned vectors include the vector data and/or metadata.
VectorsUpdate	Index Name, Project Id, Params	The Update operation updates a vector in a namespace. If a value is included, it will overwrite the previous value. If a set_metadata is included, the values of the fields specified in it will be added or overwrite the previous value.
VectorsUpsert	Index Name, Project Id, Params	The Upsert operation writes vectors into a namespace. If a new value is upserted for an existing vector id, it will overwrite the previous value.

Example UPSERT

Find below an example of UPSERT a single vector with the Id = "id1".

```
void UpsertPinecone(const String aIndexName, const String aProjectId, const std::vector<double> aVector)
{
    TsgcHTTP_API_Pinecone* oPinecone = new TsgcHTTP_API_Pinecone(NULL);
    try
    {
        oPinecone->PineconeOptions.API = "your-api-key";
        TsgcHTTPPineconeVectorUpserts* oParams = new TsgcHTTPPineconeVectorUpserts();
        try
        {
            TsgcArrayOfVectorUpsert oVectors;
            oVectors.push_back(new TsgcHTTPPineconeVectorUpsert());
            oVectors[0]->Id = "id1";
            oVectors[0]->Values = aVector;
            oParams->Vectors = oVectors;
            oPinecone->VectorsUpsert(aIndexName, aProjectId, oParams);
        }
        __finally
        {
            oParams->Free();
        }
    }
    __finally
    {
        oPinecone->Free();
    }
}
```

Example QUERY

Find below an example of QUERY a single vector.

```
void QueryPinecone(const string aIndexName, const string aProjectId, const std::vector<double>& aVector)
{
    TsgcHTTPPineconeVectorQuery* oParams = new TsgcHTTPPineconeVectorQuery();
    try
    {
        oParams->Vector = aVector;
        Pinecone.VectorsQuery(aIndexName, aProjectId, oParams);
    }
    __finally
    {
        oParams->Free();
    }
}
```

Anthropic Claude

Anthropic is an AI safety company that builds reliable, interpretable, and steerable AI systems. Their flagship model family is Claude, which excels at thoughtful dialogue, content creation, complex reasoning, coding, and more.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Anthropic** to interact with the Anthropic Claude API.

Anthropic API

The Anthropic API provides access to Claude models for building AI-powered applications. The API supports text generation, vision (image understanding), tool use (function calling), extended thinking, document/PDF processing, prompt caching, citations, web search, streaming, token counting, and message batches.

Features

- **Messages**
  - **Anthropic Messages Examples**
- **Vision**
  - **Anthropic Vision Examples**
- **Tool Use**
  - **Anthropic Tool Use Examples**
- **Models**
  - **Anthropic Models Examples**
- **Batches**
  - **Anthropic Batches Examples**
- **Extended Thinking**
  - **Anthropic Extended Thinking Examples**
- **Documents**
  - **Anthropic Documents Examples**
- **Prompt Caching**
  - **Anthropic Prompt Caching Examples**
- **Citations**
  - **Anthropic Citations Examples**
- **Web Search**
  - **Anthropic Web Search Examples**
- **Structured Outputs**
  - **Anthropic Structured Outputs Examples**
- **Files**
  - **Anthropic Files API Examples**
- **MCP Connector**
  - **Anthropic MCP Connector Examples**

Configuration

The Anthropic API uses API keys for authentication. Visit your API Keys page in the Anthropic Console to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **AnthropicOptions.ApiKey** property of the component. The **AnthropicOptions.AnthropicVersion** property specifies the API version (default: 2023-06-01).

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "YOUR_API_KEY";
```

Properties

AnthropicOptions

- **ApiKey:** The API key for authenticating with the Anthropic API.

- **AnthropicVersion:** The API version string (default: 2023-06-01).
- **BetaHeaders:** Optional beta feature headers to enable pre-release API features (e.g. files-api-2025-04-14, mcp-client-2025-11-20).
- **HttpOptions:** HTTP connection settings.
  - **ReadTimeout:** Timeout in milliseconds for reading HTTP responses. Default is 0 (no timeout).
- **LogOptions:** Logging configuration.
  - **Enabled:** When True, HTTP requests and responses are logged to a file.
  - **FileName:** The file path where log output is written.
- **RetryOptions:** Automatic retry configuration for failed API requests.
  - **Enabled:** When True, failed requests are automatically retried.
  - **Retries:** Maximum number of retry attempts.
  - **Wait:** Wait time in milliseconds between retry attempts.

Messages

Send a structured list of input messages with text and/or image content, and the model will generate the next message in the conversation.

- **CreateMessage:** Creates a message with the specified model and parameters.
  - **Model:** The model to use (e.g. claude-sonnet-4-20250514).
  - **Message:** The user message content.
  - **MaxTokens:** The maximum number of tokens to generate (required, default 4096).
- **CreateMessageWithSystem:** Creates a message with a system prompt.
  - **System:** System prompt that sets the behavior of the assistant.
- **CreateMessageStream:** Creates a message with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.
- **CreateMessageJSON:** Creates a message that returns structured JSON conforming to a provided JSON Schema.
  - **Schema:** A JSON Schema string defining the output format.
- **CreateMessageWithThinking:** Creates a message with extended thinking enabled.
  - **BudgetTokens:** Maximum token budget for thinking (minimum 1024).
- **CreateDocumentMessage:** Creates a message with a base64 document (PDF or text).
  - **DocumentBase64:** The base64-encoded document data.
  - **MediaType:** The MIME type (e.g. application/pdf).

Vision

Claude can understand images passed as base64-encoded content blocks within messages.

- **CreateVisionMessage:** Sends an image with a text prompt.
  - **ImageBase64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).
  - **Prompt:** The text prompt to accompany the image.

Tool Use

Claude can use tools (function calling) to interact with external systems. You define tools with their names, descriptions, and input schemas, and Claude will generate tool_use content blocks when it wants to call a tool.

Models

List and describe the available Claude models.

- **GetModels:** Lists all available models.
- **GetModel:** Retrieves information about a specific model.
  - **ModelId:** The ID of the model to retrieve.

Extended Thinking

Extended thinking enables Claude to reason step-by-step before responding, improving quality for complex tasks like math, coding, and analysis.

- **ThinkingType:** Set to 'enabled' to activate thinking, 'disabled' to turn it off.
- **ThinkingBudgetTokens:** Token budget for thinking (min 1024, must be less than MaxTokens).

- **CreateMessageWithAdaptiveThinking:** Creates a message with adaptive thinking, letting Claude decide the thinking depth automatically.

Documents

Claude can process PDF documents and text files sent as content blocks. Supports base64, text, and URL source types.

- **CreateDocumentMessage:** Sends a document with a text prompt.
  - **DocumentBase64:** The base64-encoded document data.
  - **MediaType:** The MIME type (application/pdf, text/plain).

Prompt Caching

Cache frequently used context (system prompts, content blocks, tool definitions) between API calls to reduce costs by up to 90% on cache reads.

- **SystemCacheControl:** Set to True on the request to cache the system prompt.
- **CacheControl:** Set to 'ephemeral' on content blocks or tool definitions.
- **BetaHeaders:** Optional beta feature headers in AnthropicOptions.

Citations

When documents are sent with citations enabled, Claude includes source references in its response. Citation types include char_location (text), page_location (PDF), content_block_location (custom content), and web_search_result_location (web search).

Web Search

Claude can search the web for real-time information using the built-in web_search tool. Other built-in tools include code_execution and computer use.

- **CreateMessageWithWebSearch:** Creates a message with web search enabled.
  - **Model:** The model to use.
  - **Message:** The user query.

Token Counting

Count the number of tokens in a message before sending it.

- **CountTokens:** Counts the number of input tokens for a message.
  - **Model:** The model to use for token counting.
  - **Message:** The message content to count tokens for.

Message Batches

The Message Batches API allows you to process large volumes of messages asynchronously.

- **ListBatches:** Lists all message batches.
- **GetBatch:** Retrieves a specific batch by ID.
- **CancelBatch:** Cancels a batch that is still processing.
- **GetBatchResults:** Retrieves the results of a completed batch.

Structured Outputs

Force Claude to return valid JSON conforming to a provided JSON Schema. Combine with the Effort parameter to control output quality vs. cost.

- **CreateMessageJSON:** Creates a message with JSON schema output.
  - **Schema:** A JSON Schema string defining the output format.
- **OutputFormatSchema:** (Request property) JSON Schema for structured output.
- **Effort:** (Request property) Controls quality vs. cost: 'low', 'medium', 'high', 'max'.
- **Strict:** (Tool property) When True, tool inputs are guaranteed to match the input_schema.

Files API

Upload, list, retrieve, download, and delete files. Uploaded files can be referenced in messages using document content blocks with file source type. Requires beta header `files-api-2025-04-14`.

- **UploadFile:** Uploads a local file. Returns file metadata.
- **ListFiles:** Lists uploaded files with pagination.
- **GetFile:** Retrieves metadata for a specific file.
- **DownloadFile:** Downloads file content.
- **DeleteFile:** Permanently deletes a file.

Request Parameters

Additional request parameters available on the Messages API.

- **ServiceTier:** Controls priority tier usage: 'auto' (default) or 'standard_only'.
- **InferenceGeo:** Controls inference geography: 'global' (default) or 'us'.
- **Container:** Container ID for reusing code execution environments across requests.
- **IsError:** (Content block property) Set True on tool_result blocks to indicate tool execution failure.
- **CacheTTL:** Extended cache time-to-live: '5m' (default) or '1h'.
- **ThinkingType:** Set to 'adaptive' to let Claude decide thinking depth automatically.

MCP Connector

Connect Claude to external MCP (Model Context Protocol) servers to access third-party tools. Requires beta header `mcp-client-2025-11-20`.

- **CreateMessageWithMCP:** Creates a message with an MCP server connection.
  - **MCPServerUrl:** HTTPS URL of the MCP server.
  - **MCPServerName:** Unique name for the server.
- **MCPServers:** (Request property) Array of MCP server definitions with ServerType, Url, Name, AuthorizationToken.
- **MCPServerName:** (Tool property) References an MCP server when ToolType is 'mcp_toolset'.

TsgcHTTP_API_Anthropic

Anthropic REST API client for Claude Messages, Files, Batches and token counting.

Introduction

Anthropic REST API client for Claude Messages, Files, Batches and token counting.

Introduction

Anthropic REST API client for Claude Messages, Files, Batches and token counting.
Configure the **AnthropicOptions** property with your API key and optional HTTP/retry/log settings, then call the typed methods such as `CreateMessage`/`CreateChatCompletion` with a request object and receive a typed response class.
For background on the provider, see the Anthropic guide.

Reference

- Properties — 5 properties available on the component.
- Methods — 7 public methods.
- Events — 2 events you can handle.

Anthropic | Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

Simple Example

Send a Hello message to Claude.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_CreateMessage("claude-sonnet-4-20250514", "Hello!"));
```

System Prompt Example

Send a message with a system prompt to control Claude's behavior.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_CreateMessageWithSystem("claude-sonnet-4-20250514",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Advanced Example

Use the typed request/response classes for full control over message parameters.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
  oRequest->Model = "claude-sonnet-4-20250514";
  oRequest->MaxTokens = 1024;
  oRequest->System = "You are a helpful assistant.";
  oRequest->Temperature = 0.7;

  TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
  oMessage->Role = "user";
  oMessage->Content = "Hello!";
  oMessages = oRequest->Messages;
  SetLength(oMessages, 1);
  oMessages[0] = oMessage;
  oRequest->Messages = oMessages;

  TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
  try {
    if (oResponse->Content.Length > 0)
      ShowMessage(oResponse->Content[0]->Text);
  } __finally {
    delete oResponse;
  }
} __finally {
  delete oMessage;
  delete oRequest;
}
```

Streaming Example

Use Server-Sent Events (SSE) to stream the response in real-time. Assign the OnHTTPAPISSE event handler to receive streaming events.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
```

```
Anthropic->OnHTTPAPISSE = OnSSEEvent;
Anthropic->_CreateMessageStream("claude-sonnet-4-20250514", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
  const String aData, bool &Cancel)
{
  // aEvent contains the event type (e.g. content_block_delta)
  // aData contains the JSON data for this event
  Memo1->Lines->Add(aData);
}
```

Anthropic | Vision

Claude can understand and analyze images. You can send images as base64-encoded data within content blocks.

Supported Image Formats

- **image/jpeg** - JPEG images
- **image/png** - PNG images
- **image/gif** - GIF images
- **image/webp** - WebP images

Simple Example

Send an image with a prompt asking Claude to describe it.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

// Load image and encode to base64
TFileStream *oStream = new TFileStream("photo.png", fmOpenRead);
try {
  TBytesStream *oBytes = new TBytesStream();
  try {
    oBytes->CopyFrom(oStream, 0);
    vBase64 = EncodeBase64(oBytes->Memory, oBytes->Size);
  } __finally {
    oBytes->Free();
  }
} __finally {
  oStream->Free();
}

ShowMessage(Anthropic->_CreateVisionMessage("claude-sonnet-4-20250514",
  "What is in this image?", vBase64, "image/png"));
```

Advanced Example

Use content blocks for more control over the image message.

```
TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages(NULL);
try {
  oRequest->Model = "claude-sonnet-4-20250514";
  oRequest->MaxTokens = 4096;

  TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message(NULL);
  oMessage->Role = "user";

  // Image content block
  TsgcAnthropicClass_Request_Content_Block *oImageBlock = new TsgcAnthropicClass_Request_Content_Block(NULL);
  oImageBlock->ContentType = "image";
  oImageBlock->MediaType = "image/jpeg";
  oImageBlock->Data = vBase64;

  // Text content block
  TsgcAnthropicClass_Request_Content_Block *oTextBlock = new TsgcAnthropicClass_Request_Content_Block(NULL);
  oTextBlock->ContentType = "text";
  oTextBlock->Text = "Describe this image in detail.";

  SetLength(oBlocks, 2);
  oBlocks[0] = oImageBlock;
  oBlocks[1] = oTextBlock;
  oMessage->ContentBlocks = oBlocks;

  SetLength(oMessages, 1);
  oMessages[0] = oMessage;
  oRequest->Messages = oMessages;

  TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
  try {
    if (Length(oResponse->Content) > 0)
```

```
      ShowMessage(oResponse->Content[0]->Text);
  } __finally {
    oResponse->Free();
  }
} __finally {
  oImageBlock->Free();
  oTextBlock->Free();
  oMessage->Free();
  oRequest->Free();
}
```

Anthropic | Tool Use

Claude supports tool use (function calling), allowing you to define tools that Claude can invoke during a conversation. When Claude decides to use a tool, it returns a **tool_use** content block. You then execute the tool and send back the result as a **tool_result** content block.

Tool Use Flow

1. Define tools with name, description, and input_schema (JSON Schema).
2. Send a message with tools defined.
3. Claude responds with a **tool_use** content block (stop_reason = 'tool_use').
4. Execute the tool with the provided input.
5. Send a new message with a **tool_result** content block containing the output.
6. Claude responds with the final answer.

Example

Define a weather tool and handle the tool use loop.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

// Step 1: Create request with tools
TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages(NULL);
try {
  oRequest->Model = "claude-sonnet-4-20250514";
  oRequest->MaxTokens = 4096;

  // Define user message
  TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message(NULL);
  oMessage->Role = "user";
  oMessage->Content = "What is the weather in San Francisco?";
  SetLength(oMessages, 1);
  oMessages[0] = oMessage;
  oRequest->Messages = oMessages;

  // Define tool
  TsgcAnthropicClass_Request_Tool *oTool = new TsgcAnthropicClass_Request_Tool(NULL);
  oTool->Name = "get_weather";
  oTool->Description = "Get the current weather in a given location";
  oTool->InputSchema =
    "{\"type\":\"object\",\"properties\":{\"location\":{\"type\":\"string\","
    "\"description\":\"The city and state\"}},\"required\":[\"location\"]}";
  SetLength(oTools, 1);
  oTools[0] = oTool;
  oRequest->Tools = oTools;

  // Step 2: Send request
  TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
  try {
    // Step 3: Check if Claude wants to use a tool
    if (oResponse->StopReason == "tool_use") {
      // Find the tool_use content block
      for (int i = 0; i < Length(oResponse->Content); i++) {
        if (oResponse->Content[i]->ContentType == "tool_use") {
          vToolUseId = oResponse->Content[i]->Id;
          vToolName = oResponse->Content[i]->Name;
          vToolInput = oResponse->Content[i]->Input;
          // Step 4: Execute your tool (get_weather) and get result
          vToolResult = "72 degrees and sunny";
          break;
        }
      }

      // Step 5: Send tool result back
      // Build new message array with assistant response + tool result
      // ... (continue the conversation with tool_result content block)
    }
  } __finally {
    oResponse->Free();
  }
} __finally {
  oMessage->Free();
```

```
  oTool->Free();
  oRequest->Free();
}
```

Anthropic | Models

List and retrieve information about available Claude models.

List Models

Lists all available Claude models.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_GetModels());
```

Get Model

Retrieves information about a specific model.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_GetModel("claude-sonnet-4-20250514"));
```

Typed Response

Use the typed response class for structured access to model data.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Response_Models *oModels = Anthropic->GetModels();
try {
  for (int i = 0; i < oModels->Data.Length; i++)
    ShowMessage(oModels->Data[i]->Id + " - " + oModels->Data[i]->DisplayName);
} __finally {
  delete oModels;
}
```

Anthropic | Message Batches

The Message Batches API allows you to process large volumes of messages asynchronously. This is ideal for tasks that don't require immediate responses, such as bulk content generation, data analysis, or batch processing workflows.

List Batches

Lists all message batches for your organization.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_ListBatches());
```

Get Batch

Retrieves information about a specific batch.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_GetBatch("batch_id_here"));
```

Cancel Batch

Cancels a batch that is still processing.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_CancelBatch("batch_id_here"));
```

Get Batch Results

Retrieves the results of a completed batch.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_GetBatchResults("batch_id_here"));
```

Typed Response

Use the typed response class for structured access to batch data.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Response_Batches *oBatches = Anthropic->ListBatches();
try {
  for (int i = 0; i < Length(oBatches->Batches); i++)
    ShowMessage(oBatches->Batches[i]->Id + " - " + oBatches->Batches[i]->ProcessingStatus);
} __finally {
  oBatches->Free();
}
```

Anthropic | Extended Thinking

Extended thinking gives Claude the ability to think through complex problems step-by-step before providing a response. When enabled, Claude creates internal reasoning (thinking blocks) that improve the quality of responses for math, coding, analysis, and other complex tasks.

Simple Example

Send a message with extended thinking enabled using the convenience method. The temperature is automatically set to 1.0 (required by the API when thinking is enabled).

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_CreateMessageWithThinking("claude-sonnet-4-20250514",
  "How many r's are in the word strawberry?", 10000));
```

Advanced Example

Use the typed request/response classes for full control. Set ThinkingType to 'enabled' and ThinkingBudgetTokens to the desired token budget (minimum 1024). The response will contain thinking and text content blocks.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
  oRequest->Model = "claude-sonnet-4-20250514";
  oRequest->MaxTokens = 16384;
  oRequest->ThinkingType = "enabled";
  oRequest->ThinkingBudgetTokens = 10000;

  TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
  oMessage->Role = "user";
  oMessage->Content = "Explain the proof that there are infinitely many primes.";
  oMessages = oRequest->Messages;
  SetLength(oMessages, 1);
  oMessages[0] = oMessage;
  oRequest->Messages = oMessages;

  TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
  try {
    for (int i = 0; i < oResponse->Content.Length; i++) {
      if (oResponse->Content[i]->ContentType == "thinking")
        ShowMessage("Thinking: " + oResponse->Content[i]->Thinking);
      else if (oResponse->Content[i]->ContentType == "text")
        ShowMessage("Response: " + oResponse->Content[i]->Text);
    }
  } __finally {
    delete oResponse;
  }
} __finally {
  delete oMessage;
  delete oRequest;
}
```

Multi-turn with Thinking

When using extended thinking in multi-turn conversations, thinking and redacted_thinking blocks from previous responses must be passed back in the conversation. Use the ContentBlocks array to include these blocks.

```
// Pass back thinking blocks from a previous response
TsgcAnthropicClass_Request_Content_Block *oBlock = new TsgcAnthropicClass_Request_Content_Block();
oBlock->ContentType = "thinking";
oBlock->Text = oPrevThinkingBlock->Thinking;   // thinking text
oBlock->Signature = oPrevThinkingBlock->Signature; // signature string

// Pass back redacted thinking blocks
```

```
TsgcAnthropicClass_Request_Content_Block *oBlock2 = new TsgcAnthropicClass_Request_Content_Block();
oBlock2->ContentType = "redacted_thinking";
oBlock2->Data = oPrevRedactedBlock->Data;
```

Properties

- **ThinkingType:** Set to 'enabled' to activate extended thinking, or 'disabled' to turn it off.
- **ThinkingBudgetTokens:** The maximum number of tokens for thinking (minimum 1024). Must be less than MaxTokens.

Notes

- Temperature must be 1.0 when thinking is enabled (the convenience method sets this automatically).
- TopK cannot be used when thinking is enabled.
- BudgetTokens must be at least 1024 and less than MaxTokens.
- Response content blocks may include: thinking, redacted_thinking, and text types.

Anthropic | Documents

Claude can process PDF documents and text files sent as content blocks within messages. Documents are sent as base64-encoded data or via URL, and Claude can analyze, summarize, and answer questions about their content.

Simple Example

Send a PDF document with a question using the convenience method.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

// Load PDF file and encode to base64
AnsiString vBase64 = sgcBase64Encode(LoadFileToBytes("document.pdf"));
ShowMessage(Anthropic->_CreateDocumentMessage("claude-sonnet-4-20250514",
  "Summarize this document.", vBase64, "application/pdf"));
```

Advanced Example with Citations

Use the typed classes to send a document with citations enabled. When citations are enabled, Claude's response will include references to specific parts of the source document.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
  oRequest->Model = "claude-sonnet-4-20250514";
  oRequest->MaxTokens = 4096;

  // Create document content block
  TsgcAnthropicClass_Request_Content_Block *oDocBlock = new TsgcAnthropicClass_Request_Content_Block();
  oDocBlock->ContentType = "document";
  oDocBlock->SourceType = "base64";
  oDocBlock->MediaType = "application/pdf";
  oDocBlock->Data = sgcBase64Encode(LoadFileToBytes("report.pdf"));
  oDocBlock->Title = "Annual Report";
  oDocBlock->CitationsEnabled = true;

  // Create text prompt block
  TsgcAnthropicClass_Request_Content_Block *oTextBlock = new TsgcAnthropicClass_Request_Content_Block();
  oTextBlock->ContentType = "text";
  oTextBlock->Text = "What are the key findings?";

  // Build message with content blocks
  TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
  oMessage->Role = "user";
  oBlocks = oMessage->ContentBlocks;
  SetLength(oBlocks, 2);
  oBlocks[0] = oDocBlock;
  oBlocks[1] = oTextBlock;
  oMessage->ContentBlocks = oBlocks;

  oMessages = oRequest->Messages;
  SetLength(oMessages, 1);
  oMessages[0] = oMessage;
  oRequest->Messages = oMessages;

  TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
  try {
    if (oResponse->Content.Length > 0)
      ShowMessage(oResponse->Content[0]->Text);
  } __finally {
    delete oResponse;
  }
} __finally {
  delete oRequest;
}
```

Properties

- **ContentType:** Set to 'document' for document content blocks.
- **SourceType:** The source type: 'base64' (default), 'text', or 'url'.
- **MediaType:** The MIME type (e.g. application/pdf, text/plain).
- **Data:** The base64-encoded document data, or URL when SourceType is 'url'.
- **Title:** Optional document title for reference.
- **Context:** Optional context or metadata about the document.
- **CitationsEnabled:** Set to True to enable source citations in the response.

Supported Formats

- **PDF:** application/pdf (max 32 MB, max 100 pages per request)
- **Plain Text:** text/plain

Anthropic | Prompt Caching

Prompt caching allows you to cache frequently used context between API calls, reducing costs by up to 90% on cache reads and improving latency. Cached content is marked with a cache_control parameter and is reused across requests within the cache TTL window.

System Prompt Caching

Cache a long system prompt to avoid re-processing it on every request. Set SystemCacheControl to True to automatically wrap the system prompt with cache_control.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages(NULL);
try {
  oRequest->Model = "claude-sonnet-4-20250514";
  oRequest->MaxTokens = 4096;
  oRequest->System = "You are an expert legal assistant... (long system prompt)";
  oRequest->SystemCacheControl = true;  // Enable caching for system prompt

  TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message(NULL);
  oMessage->Role = "user";
  oMessage->Content = "Analyze this contract clause.";
  oMessages = oRequest->Messages;
  SetLength(oMessages, 1);
  oMessages[0] = oMessage;
  oRequest->Messages = oMessages;

  TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
  try {
    // Check cache usage in response
    ShowMessage("Cache created: " +
      IntToStr(oResponse->Usage->CacheCreationInputTokens));
    ShowMessage("Cache read: " +
      IntToStr(oResponse->Usage->CacheReadInputTokens));
    ShowMessage(oResponse->Content[0]->Text);
  } __finally {
    oResponse->Free();
  }
} __finally {
  oMessage->Free();
  oRequest->Free();
}
```

Content Block Caching

Cache specific content blocks (e.g. large documents or context) by setting CacheControl to 'ephemeral' on individual content blocks.

```
TsgcAnthropicClass_Request_Content_Block *oBlock = new TsgcAnthropicClass_Request_Content_Block(NULL);
oBlock->ContentType = "text";
oBlock->Text = "(large reference text to cache)";
oBlock->CacheControl = "ephemeral";  // Mark for caching
```

Tool Definition Caching

Cache tool definitions to avoid re-processing them on every request. This is useful when you have many tool definitions that remain constant across requests.

```
TsgcAnthropicClass_Request_Tool *oTool = new TsgcAnthropicClass_Request_Tool(NULL);
oTool->Name = "search_database";
oTool->Description = "Search the database for records.";
oTool->InputSchema = "{\"type\":\"object\",\"properties\":{\"query\":{\"type\":\"string\"}}}";
oTool->CacheControl = "ephemeral";  // Cache this tool definition
```

Properties

- **SystemCacheControl:** Set to True on the request to cache the system prompt with ephemeral cache_control.
- **CacheControl:** Set to 'ephemeral' on content blocks or tool definitions to mark them for caching.
- **CacheCreationInputTokens:** (Response) Number of tokens used to create cache entries.
- **CacheReadInputTokens:** (Response) Number of tokens read from cache (cost savings).

Pricing

- **Cache writes:** 1.25x the base input token price (5-minute TTL).
- **Cache reads:** 0.1x the base input token price (90% savings).
- **Cache TTL:** 5 minutes by default. Subsequent requests refresh the TTL.

Anthropic | Citations

When documents are sent with citations enabled, Claude's response includes references back to specific parts of the source documents. This lets you verify claims and trace information to its origin.

Enabling Citations

Set CitationsEnabled to True on document content blocks. Citations must be enabled on ALL or NONE of the documents in a request.

```
TsgcAnthropicClass_Request_Content_Block *oDocBlock = new TsgcAnthropicClass_Request_Content_Block(NULL);
oDocBlock->ContentType = "document";
oDocBlock->SourceType = "base64";
oDocBlock->MediaType = "application/pdf";
oDocBlock->Data = sgcBase64Encode(LoadFileToBytes("report.pdf"));
oDocBlock->CitationsEnabled = true;  // Enable citations
```

Reading Citations from Response

Text content blocks in the response may contain a Citations array with source references.

```
TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
try {
  for (int i = 0; i < Length(oResponse->Content); i++) {
    if (oResponse->Content[i]->ContentType == "text") {
      ShowMessage(oResponse->Content[i]->Text);

      // Process citations
      for (int j = 0; j < Length(oResponse->Content[i]->Citations); j++) {
        TsgcAnthropicClass_Response_Citation *oCitation = oResponse->Content[i]->Citations[j];
        ShowMessage(Format("  Citation [%s]: \"%s\"",
          ARRAYOFCONST((oCitation->CitationType, oCitation->CitedText))));

        if (oCitation->CitationType == "page_location")
          ShowMessage(Format("    Pages %d-%d of \"%s\"",
            ARRAYOFCONST((oCitation->StartPageNumber, oCitation->EndPageNumber,
             oCitation->DocumentTitle))));
      }
    }
  }
} __finally {
  oResponse->Free();
}
```

Citation Types

- **char_location:** Character-level reference in plain text documents. Fields: DocumentIndex, DocumentTitle, StartCharIndex, EndCharIndex.
- **page_location:** Page-level reference in PDF documents. Fields: DocumentIndex, DocumentTitle, StartPageNumber, EndPageNumber.
- **content_block_location:** Block-level reference in custom content documents. Fields: DocumentIndex, DocumentTitle, StartBlockIndex, EndBlockIndex.
- **web_search_result_location:** Reference to web search results. Fields: Url, Title, EncryptedIndex.

Anthropic | Web Search

The Web Search tool enables Claude to search the web for real-time information during a conversation. This is a built-in server-side tool that Anthropic hosts and executes automatically.

Simple Example

Use the convenience method to create a message with web search enabled.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_CreateMessageWithWebSearch("claude-sonnet-4-20250514",
  "What are the latest news about Delphi programming?"));
```

Advanced Example

Use the typed classes for full control over the web search tool parameters such as MaxUses.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
  oRequest->Model = "claude-sonnet-4-20250514";
  oRequest->MaxTokens = 4096;

  // Add web search tool
  TsgcAnthropicClass_Request_Tool *oTool = new TsgcAnthropicClass_Request_Tool();
  oTool->ToolType = "web_search_20250305";
  oTool->Name = "web_search";
  oTool->MaxUses = 5;  // Max 5 searches per request
  oTools = oRequest->Tools;
  SetLength(oTools, 1);
  oTools[0] = oTool;
  oRequest->Tools = oTools;

  TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
  oMessage->Role = "user";
  oMessage->Content = "Find the current price of Bitcoin.";
  oMessages = oRequest->Messages;
  SetLength(oMessages, 1);
  oMessages[0] = oMessage;
  oRequest->Messages = oMessages;

  TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
  try {
    for (int i = 0; i < oResponse->Content.Length; i++) {
      if (oResponse->Content[i]->ContentType == "text")
        ShowMessage(oResponse->Content[i]->Text);
    }
  } __finally {
    delete oResponse;
  }
} __finally {
  delete oTool;
  delete oMessage;
  delete oRequest;
}
```

Built-in Tool Types

- **web_search_20250305:** Server-side web search. Properties: Name ('web_search'), MaxUses.
- **code_execution_20250825:** Sandboxed code execution. Properties: Name ('code_execution'), MaxUses.
- **computer_20250124:** Computer use (mouse, keyboard, screenshots). Properties: Name ('computer'), DisplayWidthPx, DisplayHeightPx.

Response Content Types

When built-in tools are used, the response may contain additional content block types:

- **server_tool_use:** Indicates Claude invoked a built-in tool. Fields: Id, Name, Input.
- **web_search_tool_result:** Contains web search results. Fields: ToolUseId, Data (raw JSON content).

Anthropic | Structured Outputs

Structured Outputs force Claude to return responses conforming to a JSON schema. This guarantees valid, parseable JSON output matching your schema definition.

Simple Example

Use the convenience method to create a message with JSON schema output.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

String vSchema = "{\"type\":\"object\",\"properties\":{\"name\":{\"type\":\"string\"},"
  "\"age\":{\"type\":\"integer\"}},\"required\":[\"name\",\"age\"],"
  "\"additionalProperties\":false}";

ShowMessage(Anthropic->_CreateMessageJSON("claude-sonnet-4-20250514",
  "Extract the name and age: John is 30 years old.", vSchema));
```

Advanced Example

Use the typed classes to combine structured output with the effort parameter.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
  oRequest->Model = "claude-sonnet-4-20250514";
  oRequest->MaxTokens = 4096;

  // JSON schema for structured output
  oRequest->OutputFormatSchema =
    "{\"type\":\"object\",\"properties\":{\"sentiment\":{\"type\":\"string\","
    "\"enum\":[\"positive\",\"negative\",\"neutral\"]},\"confidence\":"
    "{\"type\":\"number\"}},\"required\":[\"sentiment\",\"confidence\"],"
    "\"additionalProperties\":false}";

  // Set effort level (low, medium, high, max)
  oRequest->Effort = "medium";

  TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
  oMessage->Role = "user";
  oMessage->Content = "Analyze the sentiment: I love this product!";
  oMessages = oRequest->Messages;
  SetLength(oMessages, 1);
  oMessages[0] = oMessage;
  oRequest->Messages = oMessages;

  TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
  try {
    for (int i = 0; i < oResponse->Content.Length; i++) {
      if (oResponse->Content[i]->ContentType == "text")
        ShowMessage(oResponse->Content[i]->Text);
    }
  } __finally {
    delete oResponse;
  }
} __finally {
  delete oMessage;
  delete oRequest;
}
```

Strict Tool Use

Enable strict mode on tool definitions to guarantee tool inputs conform exactly to the input_schema.

```
TsgcAnthropicClass_Request_Tool *oTool = new TsgcAnthropicClass_Request_Tool();
oTool->Name = "get_weather";
```

```
oTool->Description = "Get the current weather for a location";
oTool->Strict = true;  // Guarantee schema conformance
oTool->InputSchema = "{\"type\":\"object\",\"properties\":{\"location\":"
  "{\"type\":\"string\"}},\"required\":[\"location\"],\"additionalProperties\":false}";
```

Properties

- **OutputFormatSchema:** A JSON Schema string. Claude's response text will be valid JSON conforming to this schema.
- **Effort:** Controls output quality vs. cost. Values: 'low', 'medium', 'high' (default), 'max' (Opus 4.6 only).
- **Strict:** (on tools) When True, tool inputs are guaranteed to conform to the input_schema.

Anthropic | Files API

The Files API allows you to upload, list, retrieve, download, and delete files. Uploaded files can be referenced in messages using document content blocks with file source type.
Note: The Files API requires the beta header `files-api-2025-04-14`. Set this in the **AnthropicOptions.BetaHeaders** property.

Upload a File

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
Anthropic->AnthropicOptions->BetaHeaders = "files-api-2025-04-14";

TsgcAnthropicClass_Response_File *oFile = Anthropic->UploadFile("C:\\documents\\report.pdf");
try {
  ShowMessage("File ID: " + oFile->Id);
} __finally {
  delete oFile;
}
```

List Files

```
TsgcAnthropicClass_Response_FileList *oList = Anthropic->ListFiles(50);
try {
  for (int i = 0; i < oList->Data.Length; i++)
    ShowMessage(oList->Data[i]->Id + " - " + oList->Data[i]->Filename +
      " (" + IntToStr(oList->Data[i]->SizeBytes) + " bytes)");
  ShowMessage("Has more: " + BoolToStr(oList->HasMore, true));
} __finally {
  delete oList;
}
```

Use File in Messages

Reference uploaded files using the document content block with SourceType set to 'file'.

```
TsgcAnthropicClass_Request_Content_Block *oDocBlock = new TsgcAnthropicClass_Request_Content_Block();
oDocBlock->ContentType = "document";
oDocBlock->SourceType = "file";
oDocBlock->FileId = "file_abc123";  // ID from UploadFile

TsgcAnthropicClass_Request_Content_Block *oTextBlock = new TsgcAnthropicClass_Request_Content_Block();
oTextBlock->ContentType = "text";
oTextBlock->Text = "Summarize this document.";
```

Delete a File

```
TsgcAnthropicClass_Response_File *oDeleted = Anthropic->DeleteFile("file_abc123");
try {
  ShowMessage("Deleted: " + oDeleted->Id);
} __finally {
  delete oDeleted;
}
```

API Methods

- **UploadFile:** Uploads a file from a local path. Returns file metadata (Id, Filename, MimeType, SizeBytes).
- **ListFiles:** Lists all uploaded files with pagination support (Limit, AfterId, BeforeId).
- **GetFile:** Retrieves metadata for a specific file by ID.
- **DownloadFile:** Downloads the content of a file (only files created by code execution tool are downloadable).

- **DeleteFile:** Permanently deletes a file by ID.

Anthropic | MCP Connector

The MCP (Model Context Protocol) connector allows Claude to access tools from external MCP servers. This enables integration with third-party services and custom tool providers.
Note: MCP connector requires the beta header `mcp-client-2025-11-20`. Set this in the **AnthropicOptions.BetaHeaders** property.

Simple Example

Use the convenience method to create a message with an MCP server.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
Anthropic->AnthropicOptions->BetaHeaders = "mcp-client-2025-11-20";

ShowMessage(Anthropic->_CreateMessageWithMCP("claude-sonnet-4-20250514",
  "What tools are available?",
  "https://my-mcp-server.example.com/sse",
  "my-mcp-server"));
```

Advanced Example

Use the typed classes for full control over MCP server configuration, including authentication.

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
Anthropic->AnthropicOptions->BetaHeaders = "mcp-client-2025-11-20";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
  oRequest->Model = "claude-sonnet-4-20250514";
  oRequest->MaxTokens = 4096;

  // Configure MCP server
  TsgcAnthropicClass_Request_MCPServer *oServer = new TsgcAnthropicClass_Request_MCPServer();
  oServer->ServerType = "url";
  oServer->Url = "https://my-mcp-server.example.com/sse";
  oServer->Name = "my-server";
  oServer->AuthorizationToken = "OAUTH_TOKEN";  // Optional auth
  oServers = oRequest->MCPServers;
  SetLength(oServers, 1);
  oServers[0] = oServer;
  oRequest->MCPServers = oServers;

  // Add MCP toolset
  TsgcAnthropicClass_Request_Tool *oTool = new TsgcAnthropicClass_Request_Tool();
  oTool->ToolType = "mcp_toolset";
  oTool->MCPServerName = "my-server";
  oTools = oRequest->Tools;
  SetLength(oTools, 1);
  oTools[0] = oTool;
  oRequest->Tools = oTools;

  // Add user message
  TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
  oMessage->Role = "user";
  oMessage->Content = "Search for recent news about AI.";
  oMessages = oRequest->Messages;
  SetLength(oMessages, 1);
  oMessages[0] = oMessage;
  oRequest->Messages = oMessages;

  TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
  try {
    for (int i = 0; i < oResponse->Content.Length; i++) {
      if (oResponse->Content[i]->ContentType == "text")
        ShowMessage(oResponse->Content[i]->Text);
      else if (oResponse->Content[i]->ContentType == "mcp_tool_use")
        ShowMessage("MCP tool call: " + oResponse->Content[i]->Name +
          " on " + oResponse->Content[i]->ServerName);
    }
  } __finally {
```

```
    delete oResponse;
  }
} __finally {
  delete oServer;
  delete oTool;
  delete oMessage;
  delete oRequest;
}
```

MCP Server Properties

- **ServerType:** Currently only 'url' is supported.
- **Url:** The HTTPS URL of the MCP server.
- **Name:** Unique identifier for this server. Must match the MCPServerName in the toolset.
- **AuthorizationToken:** Optional OAuth Bearer token for authenticated servers.

Response Content Types

- **mcp_tool_use:** Claude invoked an MCP tool. Fields: Id, Name, ServerName, Input.
- **mcp_tool_result:** Result from an MCP tool execution. Fields: ToolUseId, IsError, Data (content).

Gemini

Google Gemini is a family of multimodal AI models developed by Google DeepMind. Gemini models support text generation, vision, structured outputs, embeddings, and tool use, offering powerful capabilities for building AI-powered applications.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Gemini** to interact with the Gemini API.

Gemini API

The Gemini API provides access to Google Gemini models for building AI-powered applications. The API supports content generation, vision (image understanding), structured JSON outputs, streaming, token counting, embeddings, tool use (function calling), and model listing.

Features

- **Messages**
  - **Gemini Messages Examples**
- **Vision**
  - **Gemini Vision Examples**
- **Models**
  - **Gemini Models Examples**
- **Structured Outputs**
  - **Gemini Structured Outputs Examples**
- **Token Counting**
  - **Gemini Token Counting Examples**
- **Embeddings**
  - **Gemini Embeddings Examples**
- **Tool Use**
  - **Gemini Tool Use Examples**

Configuration

The Gemini API uses API keys for authentication. Visit your API Keys page in Google AI Studio to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **GeminiOptions.ApiKey** property of the component.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "YOUR_API_KEY";
```

Messages

Send content to a Gemini model and receive generated responses. The model generates the next message based on the provided input.

- **_CreateContent:** Creates content with the specified model and user prompt.
  - **Model:** The model to use (e.g. gemini-2.0-flash).
  - **Message:** The user message content.
  - **MaxOutputTokens:** Maximum number of tokens to generate.
- **_CreateContentWithSystem:** Creates content with a system instruction.
  - **Model:** The model to use.
  - **System:** System instruction that sets the behavior of the model.
  - **Message:** The user message content.
  - **MaxOutputTokens:** Maximum number of tokens to generate.
- **_CreateContentStream:** Creates content with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.

Vision

Gemini models can understand images passed as base64-encoded content along with text prompts.

- **_CreateVisionContent:** Sends an image with a text prompt.
  - **Model:** The model to use.
  - **Prompt:** The text prompt to accompany the image.
  - **Base64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).
  - **MaxOutputTokens:** Maximum number of tokens to generate.

Structured Outputs

Generate structured JSON output from a Gemini model by providing a JSON schema that defines the expected response format.

- **_CreateContentJSON:** Creates content with structured JSON output.
  - **Model:** The model to use.
  - **Message:** The user message content.
  - **Schema:** A JSON schema defining the expected output structure.
  - **MaxOutputTokens:** Maximum number of tokens to generate.

Models

List and retrieve details about available Gemini models.

- **_GetModels:** Lists all available models.
- **_GetModel:** Gets details for a specific model.
  - **ModelId:** The identifier of the model to retrieve.

Token Counting

Count the number of tokens in a message before sending it to a model.

- **_CountTokens:** Counts tokens for a message.
  - **Model:** The model to use for tokenization.
  - **Message:** The text content to count tokens for.

Embeddings

Generate vector embeddings for text content using Gemini models.

- **_EmbedContent:** Generates embeddings for text.
  - **Model:** The model to use for embedding generation.
  - **Text:** The text content to generate embeddings for.

TsgcHTTP_API_Gemini

Google Gemini REST API client for content generation, token counting and embeddings.

Introduction

Google Gemini REST API client for content generation, token counting and embeddings.

Introduction

Google Gemini REST API client for content generation, token counting and embeddings.
Configure the **GeminiOptions** property with your API key and optional HTTP/retry/log settings, then call the typed methods such as `CreateMessage`/`CreateChatCompletion` with a request object and receive a typed response class.
For background on the provider, see the Gemini guide.

Reference

- Properties — 5 properties available on the component.
- Methods — 3 public methods.
- Events — 2 events you can handle.

DeepSeek

DeepSeek is a Chinese AI company focused on building powerful open-source language models. Their models excel at coding, reasoning, and general-purpose tasks, offering strong performance at competitive pricing.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_DeepSeek** to interact with the DeepSeek API.

DeepSeek API

The DeepSeek API provides access to DeepSeek models for building AI-powered applications. The API supports text generation, vision (image understanding), streaming, and model listing. The API follows an OpenAI-compatible format, making it easy to integrate.

Features

- **Messages**
  - **DeepSeek Messages Examples**
- **Vision**
  - **DeepSeek Vision Examples**
- **Models**
  - **DeepSeek Models Examples**

Configuration

The DeepSeek API uses API keys for authentication. Visit your API Keys page in the DeepSeek Platform to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **DeepSeekOptions.ApiKey** property of the component.

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "YOUR_API_KEY";
```

Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

- **_CreateMessage:** Creates a message with the specified model and user prompt.
  - **Model:** The model to use (e.g. deepseek-chat).
  - **Message:** The user message content.
- **_CreateMessageWithSystem:** Creates a message with a system prompt.
  - **System:** System prompt that sets the behavior of the assistant.
- **_CreateMessageStream:** Creates a message with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.

Vision

DeepSeek can understand images passed as base64-encoded content within messages.

- **_CreateVisionMessage:** Sends an image with a text prompt.
  - **ImageBase64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).
  - **Prompt:** The text prompt to accompany the image.

Models

List the available DeepSeek models.

- **_GetModels:** Lists all available models.

TsgcHTTP_API_DeepSeek

DeepSeek REST API client for Chat Completions.

Introduction

DeepSeek REST API client for Chat Completions.

Introduction

DeepSeek REST API client for Chat Completions.
Configure the **DeepSeekOptions** property with your API key and optional HTTP/retry/log settings, then call the typed methods such as `CreateMessage`/`CreateChatCompletion` with a request object and receive a typed response class.
For background on the provider, see the DeepSeek guide.

Reference

- Properties — 5 properties available on the component.
- Methods — 1 public methods.
- Events — 2 events you can handle.

DeepSeek | Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

Simple Example

Send a Hello message to DeepSeek.

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "API_KEY";
ShowMessage(DeepSeek->_CreateMessage("deepseek-chat", "Hello!"));
```

System Prompt Example

Send a message with a system prompt to control DeepSeek's behavior.

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "API_KEY";
ShowMessage(DeepSeek->_CreateMessageWithSystem("deepseek-chat",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Streaming Example

Use Server-Sent Events (SSE) to stream the response in real-time. Assign the OnHTTPAPISSE event handler to receive streaming events.

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "API_KEY";
DeepSeek->OnHTTPAPISSE = OnSSEEvent;
DeepSeek->_CreateMessageStream("deepseek-chat", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
  const String aData, bool &Cancel)
{
  // aEvent contains the event type
  // aData contains the JSON data for this event
  Memo1->Lines->Add(aData);
}
```

DeepSeek | Vision

DeepSeek can understand and analyze images. You can send images as base64-encoded data within content blocks.

Vision Example

Send an image with a prompt asking DeepSeek to describe it.

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "API_KEY";

// Load image and encode to base64
TFileStream *oStream = new TFileStream("photo.jpg", fmOpenRead);
try {
  TBytesStream *oBytes = new TBytesStream();
  try {
    oBytes->CopyFrom(oStream, 0);
    vImageBase64 = EncodeBase64(oBytes->Memory, oBytes->Size);
  } __finally {
    oBytes->Free();
  }
} __finally {
  oStream->Free();
}

ShowMessage(DeepSeek->_CreateVisionMessage("deepseek-chat",
  "Describe this image", vImageBase64, "image/jpeg"));
```

DeepSeek | Models

List the available DeepSeek models.

List Models

Lists all available DeepSeek models.

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "API_KEY";
ShowMessage(DeepSeek->_GetModels());
```

Gemini

Google Gemini is a family of multimodal AI models developed by Google DeepMind. Gemini models support text generation, vision, structured outputs, embeddings, and tool use, offering powerful capabilities for building AI-powered applications.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Gemini** to interact with the Gemini API.

Gemini API

The Gemini API provides access to Google Gemini models for building AI-powered applications. The API supports content generation, vision (image understanding), structured JSON outputs, streaming, token counting, embeddings, tool use (function calling), and model listing.

Features

- **Messages**
  - **Gemini Messages Examples**
- **Vision**
  - **Gemini Vision Examples**
- **Models**
  - **Gemini Models Examples**
- **Structured Outputs**
  - **Gemini Structured Outputs Examples**
- **Token Counting**
  - **Gemini Token Counting Examples**
- **Embeddings**
  - **Gemini Embeddings Examples**
- **Tool Use**
  - **Gemini Tool Use Examples**

Configuration

The Gemini API uses API keys for authentication. Visit your API Keys page in Google AI Studio to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **GeminiOptions.ApiKey** property of the component.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "YOUR_API_KEY";
```

Messages

Send content to a Gemini model and receive generated responses. The model generates the next message based on the provided input.

- **_CreateContent:** Creates content with the specified model and user prompt.
  - **Model:** The model to use (e.g. gemini-2.0-flash).
  - **Message:** The user message content.
  - **MaxOutputTokens:** Maximum number of tokens to generate.
- **_CreateContentWithSystem:** Creates content with a system instruction.
  - **Model:** The model to use.
  - **System:** System instruction that sets the behavior of the model.
  - **Message:** The user message content.
  - **MaxOutputTokens:** Maximum number of tokens to generate.
- **_CreateContentStream:** Creates content with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.

Vision

Gemini models can understand images passed as base64-encoded content along with text prompts.

- **_CreateVisionContent:** Sends an image with a text prompt.
  - **Model:** The model to use.
  - **Prompt:** The text prompt to accompany the image.
  - **Base64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).
  - **MaxOutputTokens:** Maximum number of tokens to generate.

Structured Outputs

Generate structured JSON output from a Gemini model by providing a JSON schema that defines the expected response format.

- **_CreateContentJSON:** Creates content with structured JSON output.
  - **Model:** The model to use.
  - **Message:** The user message content.
  - **Schema:** A JSON schema defining the expected output structure.
  - **MaxOutputTokens:** Maximum number of tokens to generate.

Models

List and retrieve details about available Gemini models.

- **_GetModels:** Lists all available models.
- **_GetModel:** Gets details for a specific model.
  - **ModelId:** The identifier of the model to retrieve.

Token Counting

Count the number of tokens in a message before sending it to a model.

- **_CountTokens:** Counts tokens for a message.
  - **Model:** The model to use for tokenization.
  - **Message:** The text content to count tokens for.

Embeddings

Generate vector embeddings for text content using Gemini models.

- **_EmbedContent:** Generates embeddings for text.
  - **Model:** The model to use for embedding generation.
  - **Text:** The text content to generate embeddings for.

Gemini | Messages

Send a prompt to a Gemini model and receive generated content. The Gemini API uses the generateContent endpoint to process text input and return model responses.

Simple Example

Send a Hello message to Gemini.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_CreateContent("gemini-2.0-flash", "Hello!"));
```

System Instruction Example

Send a message with a system instruction to control Gemini's behavior.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_CreateContentWithSystem("gemini-2.0-flash",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Streaming Example

Use Server-Sent Events (SSE) to stream the response in real-time. Assign the OnHTTPAPISSE event handler to receive streaming events.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
Gemini->OnHTTPAPISSE = OnSSEEvent;
Gemini->_CreateContentStream("gemini-2.0-flash", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
  const String aData, bool &Cancel)
{
  Memo1->Lines->Add(aData);
}
```

Advanced Example

Use the typed request/response classes for full control over content generation parameters.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

TsgcGeminiClass_Request_GenerateContent *oRequest = new TsgcGeminiClass_Request_GenerateContent();
try {
  oRequest->Model = "gemini-2.0-flash";
  oRequest->MaxOutputTokens = 4096;
  oRequest->Temperature = 0.7;
  oRequest->SystemInstruction = "You are a creative writer.";

  SetLength(oContents, 1);
  oContents[0] = new TsgcGeminiClass_Request_Content();
  oContents[0]->Role = "user";
  SetLength(oParts, 1);
  oParts[0] = new TsgcGeminiClass_Request_Part();
  oParts[0]->Text = "Write a haiku about programming.";
  oContents[0]->Parts = oParts;
  oRequest->Contents = oContents;

  TsgcGeminiClass_Response_GenerateContent *oResponse = Gemini->CreateContent(oRequest);
  try {
```

```
    if (Length(oResponse->Candidates) > 0)
      if (Length(oResponse->Candidates[0]->Parts) > 0)
        ShowMessage(oResponse->Candidates[0]->Parts[0]->Text);
  } __finally {
    delete oResponse;
  }
} __finally {
  delete oParts[0];
  delete oContents[0];
  delete oRequest;
}
```

Gemini | Vision

Gemini can understand and analyze images. You can send images as base64-encoded data within content parts.

Vision Example

Send a base64-encoded image to Gemini for analysis.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

// Load image and encode to base64
TFileStream *oStream = new TFileStream("photo.jpg", fmOpenRead);
try {
  TBytesStream *oBytes = new TBytesStream();
  try {
    oBytes->CopyFrom(oStream, 0);
    vImageBase64 = EncodeBase64(oBytes->Memory, oBytes->Size);
  } __finally {
    oBytes->Free();
  }
} __finally {
  oStream->Free();
}

ShowMessage(Gemini->_CreateVisionContent("gemini-2.0-flash",
  "Describe this image", vImageBase64, "image/jpeg"));
```

Gemini | Models

List and retrieve information about available Gemini models.

List Models

Lists all available models.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_GetModels());
```

Get Model

Get details for a specific model.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_GetModel("gemini-2.0-flash"));
```

Advanced Example

Use the typed response classes for full control over model data.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

TsgcGeminiClass_Response_Models *oModels = Gemini->GetModels();
try {
  for (int i = 0; i < oModels->Models.Length; i++)
    ShowMessage(oModels->Models[i]->DisplayName + " - " + oModels->Models[i]->Description);
} __finally {
  delete oModels;
}
```

Gemini | Structured Outputs

Gemini can return responses in structured JSON format conforming to a schema you define. This guarantees valid, parseable JSON output matching your schema definition.

Simple Example

Use the convenience method to create content with JSON schema output.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

String vSchema = "{\"type\":\"object\",\"properties\":{\"name\":{\"type\":\"string\"},"
  "\"age\":{\"type\":\"integer\"}},\"required\":[\"name\",\"age\"]}";

ShowMessage(Gemini->_CreateContentJSON("gemini-2.0-flash",
  "Extract the name and age: John is 30 years old.", vSchema));
```

Advanced Example

Use the typed classes to configure structured output with ResponseMimeType and ResponseSchema.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

TsgcGeminiClass_Request_GenerateContent *oRequest = new TsgcGeminiClass_Request_GenerateContent();
try {
  oRequest->Model = "gemini-2.0-flash";
  oRequest->MaxOutputTokens = 4096;
  oRequest->ResponseMimeType = "application/json";
  oRequest->ResponseSchema =
    "{\"type\":\"object\",\"properties\":{\"sentiment\":{\"type\":\"string\"},"
    "\"confidence\":{\"type\":\"number\"}},\"required\":[\"sentiment\",\"confidence\"]}";

  SetLength(oContents, 1);
  oContents[0] = new TsgcGeminiClass_Request_Content();
  oContents[0]->Role = "user";
  SetLength(oParts, 1);
  oParts[0] = new TsgcGeminiClass_Request_Part();
  oParts[0]->Text = "Analyze the sentiment: I love this product!";
  oContents[0]->Parts = oParts;
  oRequest->Contents = oContents;

  TsgcGeminiClass_Response_GenerateContent *oResponse = Gemini->CreateContent(oRequest);
  try {
    if (Length(oResponse->Candidates) > 0)
      if (Length(oResponse->Candidates[0]->Parts) > 0)
        ShowMessage(oResponse->Candidates[0]->Parts[0]->Text);
  } __finally {
    delete oResponse;
  }
} __finally {
  delete oParts[0];
  delete oContents[0];
  delete oRequest;
}
```

Properties

- **ResponseMimeType:** Set to 'application/json' to enable structured JSON output.
- **ResponseSchema:** A JSON Schema string defining the expected output structure.

Gemini | Token Counting

Count the number of tokens in a message before sending it to the API. This helps estimate costs and ensure messages fit within model limits.

Simple Example

Count tokens for a message.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_CountTokens("gemini-2.0-flash", "Hello, how are you?"));
```

Advanced Example

Use the typed request/response classes for full control over token counting parameters.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

TsgcGeminiClass_Request_CountTokens *oRequest = new TsgcGeminiClass_Request_CountTokens();
try {
  oRequest->Model = "gemini-2.0-flash";
  SetLength(oContents, 1);
  oContents[0] = new TsgcGeminiClass_Request_Content();
  oContents[0]->Role = "user";
  SetLength(oParts, 1);
  oParts[0] = new TsgcGeminiClass_Request_Part();
  oParts[0]->Text = "Explain quantum computing in simple terms.";
  oContents[0]->Parts = oParts;
  oRequest->Contents = oContents;

  TsgcGeminiClass_Response_CountTokens *oResponse = Gemini->CountTokens(oRequest);
  try {
    ShowMessage("Total tokens: " + IntToStr(oResponse->TotalTokens));
  } __finally {
    delete oResponse;
  }
} __finally {
  delete oParts[0];
  delete oContents[0];
  delete oRequest;
}
```

Gemini | Embeddings

Generate vector embeddings for text content. Embeddings capture the semantic meaning of text and can be used for similarity search, clustering, and other NLP tasks.

Simple Example

Generate embeddings for text.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_EmbedContent("text-embedding-004", "Hello world"));
```

Advanced Example

Use the typed response classes for full control over embedding data.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

TsgcGeminiClass_Response_Embedding *oEmbedding = Gemini->EmbedContent("text-embedding-004", "Hello world");
try {
  ShowMessage("Dimensions: " + IntToStr(oEmbedding->Values.Length));
  for (int i = 0; i < oEmbedding->Values.Length; i++)
    ShowMessage(FloatToStr(oEmbedding->Values[i]));
} __finally {
  delete oEmbedding;
}
```

Gemini | Tool Use

Gemini supports function calling (tool use), allowing you to define functions that the model can invoke during a conversation. When Gemini decides to call a function, it returns a functionCall part in its response. You then execute the function and send the result back.

Function Calling Flow

1. Define function declarations with name, description, and parameters (JSON Schema).
2. Send a content generation request with function declarations.
3. Gemini responds with a **functionCall** part containing the function name and arguments.
4. Execute the function with the provided arguments.
5. Send a new request with a **functionResponse** part containing the result.
6. Gemini responds with the final answer.

Example

Define a weather function and handle function calling.

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

// Step 1: Create request with function declarations
TsgcGeminiClass_Request_GenerateContent *oRequest = new TsgcGeminiClass_Request_GenerateContent();
try {
  oRequest->Model = "gemini-2.0-flash";
  oRequest->MaxOutputTokens = 4096;

  // Define user message
  SetLength(oContents, 1);
  oContents[0] = new TsgcGeminiClass_Request_Content();
  oContents[0]->Role = "user";
  SetLength(oParts, 1);
  oParts[0] = new TsgcGeminiClass_Request_Part();
  oParts[0]->Text = "What is the weather in San Francisco?";
  oContents[0]->Parts = oParts;
  oRequest->Contents = oContents;

  // Define function
  SetLength(oFunctions, 1);
  oFunctions[0] = new TsgcGeminiClass_Request_FunctionDeclaration();
  oFunctions[0]->Name = "get_weather";
  oFunctions[0]->Description = "Get the current weather in a given location";
  oFunctions[0]->Parameters =
    "{\"type\":\"object\",\"properties\":{\"location\":{\"type\":\"string\","
    "\"description\":\"The city and state\"}},\"required\":[\"location\"]}";
  oRequest->FunctionDeclarations = oFunctions;

  // Step 2: Send request
  TsgcGeminiClass_Response_GenerateContent *oResponse = Gemini->CreateContent(oRequest);
  try {
    // Step 3: Check for function call
    if (Length(oResponse->Candidates) > 0) {
      if (Length(oResponse->Candidates[0]->Parts) > 0) {
        if (oResponse->Candidates[0]->Parts[0]->FunctionCallName != "") {
          vFunctionName = oResponse->Candidates[0]->Parts[0]->FunctionCallName;
          vFunctionArgs = oResponse->Candidates[0]->Parts[0]->FunctionCallArgs;
          // Step 4: Execute function and get result
          vResult = "72 degrees and sunny";
        }
      }
    }
  } __finally {
    delete oResponse;
  }
} __finally {
  delete oParts[0];
  delete oContents[0];
  delete oFunctions[0];
  delete oRequest;
}
```

Properties

- **FunctionDeclarations:** Array of TsgcGeminiClass_Request_FunctionDeclaration defining available functions.
    - **Name:** The function name.
    - **Description:** A description of what the function does.
    - **Parameters:** JSON Schema string defining the function parameters.
- **ToolChoice:** Controls function calling behavior. Set the mode (e.g. 'AUTO', 'ANY', 'NONE').

Ollama

Ollama is an open-source tool for running large language models locally. It supports a wide range of models including Llama, Mistral, Gemma, Phi, and many others, enabling local AI inference without requiring cloud API access. The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Ollama** to interact with the Ollama API.

Ollama API

The Ollama API provides access to locally running models for AI-powered applications. The API supports text generation, streaming, model management, and embeddings. No API key is required by default since models run locally. The host and port are configurable.

Features

- **Messages**
  - **Ollama Messages Examples**
- **Models**
  - **Ollama Models Examples**
- **Embeddings**
  - **Ollama Embeddings Examples**

Configuration

Ollama runs locally and by default listens on **http://localhost:11434**. Configure the host in the **OllamaOptions.Host** property. An API key is optional and only needed if you have configured authentication on your Ollama instance.

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
```

Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

- **_CreateMessage:** Creates a message with the specified model and user prompt.
  - **Model:** The model to use (e.g. llama3).
  - **Message:** The user message content.
- **_CreateMessageWithSystem:** Creates a message with a system prompt.
  - **System:** System prompt that sets the behavior of the assistant.
- **_CreateMessageStream:** Creates a message with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.

Models

Manage and query locally available models.

- **_GetModels:** Lists all available models.
- **_GetTags:** Lists all locally available model tags.
- **_ShowModel:** Retrieves detailed information about a specific model.
  - **Model:** The name of the model to query.
- **_PullModel:** Downloads a model from the Ollama model library.
  - **Model:** The name of the model to pull.
- **_DeleteModel:** Deletes a locally available model.
  - **Model:** The name of the model to delete.

Embeddings

Generate vector embeddings from text input using locally running models.

- **_CreateEmbeddings:** Generates embeddings for the given text.
  - **Model:** The model to use for embeddings (e.g. llama3).
  - **Input:** The text to generate embeddings for.

TsgcHTTP_API_Ollama

Ollama REST API client for local Chat Completions and Embeddings.

Introduction

Ollama REST API client for local Chat Completions and Embeddings.

Introduction

Ollama REST API client for local Chat Completions and Embeddings.
Configure the **OllamaOptions** property with your API key and optional HTTP/retry/log settings, then call the typed methods such as `CreateMessage`/`CreateChatCompletion` with a request object and receive a typed response class.
For background on the provider, see the Ollama guide.

Reference

- Properties — 5 properties available on the component.
- Methods — 2 public methods.
- Events — 2 events you can handle.

Ollama | Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

Simple Example

Send a Hello message to a local Ollama model.

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_CreateMessage("llama3", "Hello!"));
```

System Prompt Example

Send a message with a system prompt to control the model's behavior.

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_CreateMessageWithSystem("llama3",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Streaming Example

Use Server-Sent Events (SSE) to stream the response in real-time. Assign the OnHTTPAPISSE event handler to receive streaming events.

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
Ollama->OnHTTPAPISSE = OnSSEEvent;
Ollama->_CreateMessageStream("llama3", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
  const String aData, bool &Cancel)
{
  // aEvent contains the event type
  // aData contains the JSON data for this event
  Memo1->Lines->Add(aData);
}
```

Ollama | Models

Manage and query locally available Ollama models.

List Models

Lists all available models.

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_GetModels());
```

Get Tags

Lists all locally available model tags.

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_GetTags());
```

Show Model

Retrieves detailed information about a specific model.

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_ShowModel("llama3"));
```

Pull Model

Downloads a model from the Ollama model library.

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_PullModel("llama3"));
```

Ollama | Embeddings

Generate vector embeddings from text input using locally running Ollama models. Embeddings can be used for semantic search, clustering, and other NLP tasks.

Create Embeddings

Generates embeddings for the given text input.

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_CreateEmbeddings("llama3", "Hello world"));
```

Grok

Grok is a conversational AI assistant developed by xAI, designed to provide helpful and informative responses with real-time knowledge. Grok models are known for their strong reasoning capabilities and up-to-date information access.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Grok** to interact with the Grok API.

Grok API

The Grok API provides access to Grok models for building AI-powered applications. The API supports text generation, vision (image understanding), streaming, and model listing.

Features

- **Messages**
  - **Grok Messages Examples**
- **Vision**
  - **Grok Vision Examples**
- **Models**
  - **Grok Models Examples**

Configuration

The Grok API uses API keys for authentication. Visit your xAI Console to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **GrokOptions.ApiKey** property of the component.

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "YOUR_API_KEY";
```

Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

- **_CreateMessage:** Creates a message with the specified model and user prompt.
  - **Model:** The model to use (e.g. grok-3).
  - **Message:** The user message content.
- **_CreateMessageWithSystem:** Creates a message with a system prompt.
  - **System:** System prompt that sets the behavior of the assistant.
- **_CreateMessageStream:** Creates a message with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.

Vision

Grok can understand images passed as base64-encoded content within messages.

- **_CreateVisionMessage:** Sends an image with a text prompt.
  - **ImageBase64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).
  - **Prompt:** The text prompt to accompany the image.

Models

List the available Grok models.

- **_GetModels:** Lists all available models.

TsgcHTTP_API_Grok

TsgcHTTP_API_Grok component.

Introduction

xAI Grok REST API client for Chat Completions and model listing.

Introduction

xAI Grok REST API client for Chat Completions and model listing.
Configure the **GrokOptions** property with your API key and optional HTTP/retry/log settings, then call the typed methods such as `CreateMessage`/`CreateChatCompletion` with a request object and receive a typed response class.
For background on the provider, see the Grok guide.

Reference

- Properties — 7 properties available on the component.
- Methods — 1 public methods.
- Events — 0 events you can handle.

Grok | Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

Simple Example

Send a Hello message to Grok.

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "API_KEY";
ShowMessage(Grok->_CreateMessage("grok-3", "Hello!"));
```

System Prompt Example

Send a message with a system prompt to control Grok's behavior.

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "API_KEY";
ShowMessage(Grok->_CreateMessageWithSystem("grok-3",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Streaming Example

Use Server-Sent Events (SSE) to stream the response in real-time. Assign the OnHTTPAPISSE event handler to receive streaming events.

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "API_KEY";
Grok->OnHTTPAPISSE = OnSSEEvent;
Grok->_CreateMessageStream("grok-3", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
  const String aData, bool &Cancel)
{
  Memo1->Lines->Add(aData);
}
```

Grok | Vision

Grok can understand images passed as base64-encoded content within messages. Use the vision endpoint to send an image along with a text prompt and receive a description or analysis.

Vision Example

Send a base64-encoded image to Grok for analysis.

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "API_KEY";
ShowMessage(Grok->_CreateVisionMessage("grok-2-vision",
  "Describe this image.", vBase64, "image/jpeg"));
```

Grok | Models

List the available Grok models and their basic information.

List Models Example

Retrieve the list of all available Grok models.

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "API_KEY";
ShowMessage(Grok->_GetModels);
```

Mistral AI

Mistral AI is a French artificial intelligence company that develops efficient and powerful language models. Their models are known for strong performance across reasoning, coding, and multilingual tasks, while maintaining competitive efficiency.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Mistral** to interact with the Mistral AI API.

Mistral API

The Mistral API provides access to Mistral models for building AI-powered applications. The API supports text generation, vision (image understanding), streaming, embeddings, JSON mode, and model listing.

Features

- **Messages**
  - **Mistral Messages Examples**
- **Vision**
  - **Mistral Vision Examples**
- **Models**
  - **Mistral Models Examples**
- **Embeddings**
  - **Mistral Embeddings Examples**

Configuration

The Mistral API uses API keys for authentication. Visit your API Keys page in the Mistral Console to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **MistralOptions.ApiKey** property of the component.

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "YOUR_API_KEY";
```

Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

- **_CreateMessage:** Creates a message with the specified model and user prompt.
  - **Model:** The model to use (e.g. mistral-large-latest).
  - **Message:** The user message content.
- **_CreateMessageWithSystem:** Creates a message with a system prompt.
  - **System:** System prompt that sets the behavior of the assistant.
- **_CreateMessageStream:** Creates a message with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.
- **_CreateMessageJSON:** Creates a message with JSON mode enabled. The model will return a valid JSON response.
  - **Model:** The model to use.
  - **Message:** The user message content.

Vision

Mistral can understand images passed as base64-encoded content within messages.

- **_CreateVisionMessage:** Sends an image with a text prompt.
  - **ImageBase64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).

- **Prompt:** The text prompt to accompany the image.

Models

List the available Mistral models.

- **_GetModels:** Lists all available models.

Embeddings

Get a vector representation of a given input that can be used for semantic search, clustering, and other machine learning tasks.

- **_CreateEmbeddings:** Creates an embedding vector representing the input text.
  - **Model:** The model to use (e.g. mistral-embed).
  - **Input:** Input text to get embeddings for.

TsgcHTTP_API_Mistral

Mistral REST API client for Chat Completions and Embeddings.

Introduction

Mistral REST API client for Chat Completions and Embeddings.

Introduction

Mistral REST API client for Chat Completions and Embeddings.
Configure the **MistralOptions** property with your API key and optional HTTP/retry/log settings, then call the typed methods such as `CreateMessage`/`CreateChatCompletion` with a request object and receive a typed response class.
For background on the provider, see the Mistral guide.

Reference

- Properties — 5 properties available on the component.
- Methods — 2 public methods.
- Events — 2 events you can handle.

Mistral | Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

Simple Example

Send a Hello message to Mistral.

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_CreateMessage("mistral-large-latest", "Hello!"));
```

System Prompt Example

Send a message with a system prompt to control Mistral's behavior.

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_CreateMessageWithSystem("mistral-large-latest",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Streaming Example

Use Server-Sent Events (SSE) to stream the response in real-time. Assign the OnHTTPAPISSE event handler to receive streaming events.

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
Mistral->OnHTTPAPISSE = OnSSEEvent;
Mistral->_CreateMessageStream("mistral-large-latest", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
  const String aData, bool &Cancel)
{
  Memo1->Lines->Add(aData);
}
```

JSON Mode Example

Request a JSON-formatted response from Mistral.

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_CreateMessageJSON("mistral-large-latest",
  "List 3 colors as JSON"));
```

Mistral | Vision

Mistral can understand images passed as base64-encoded content within messages. Use the vision endpoint to send an image along with a text prompt and receive a description or analysis.

Vision Example

Send a base64-encoded image to Mistral for analysis.

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_CreateVisionMessage("mistral-large-latest",
  "Describe this image.", vBase64, "image/jpeg"));
```

Mistral | Models

List the available Mistral models and their basic information.

List Models Example

Retrieve the list of all available Mistral models.

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_GetModels);
```

Mistral | Embeddings

Get a vector representation of a given input that can be used for semantic search, clustering, and other machine learning tasks.

Embeddings Example

Create an embedding vector for a text input.

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_CreateEmbeddings("mistral-embed", "Hello world"));
```

Cryptography

sgcWebSockets includes **sgcCrypto**, a general purpose cryptography library written entirely in pure Pascal. It has no dependency on OpenSSL or any other external library, and it compiles unchanged from Delphi 7 through Delphi 13, on Win32, Win64, Linux64, macOS and iOS.
The library covers hashing and message authentication, password hashing and key derivation, symmetric encryption, classical public key cryptography, post-quantum cryptography, X.509 certificates, one time passwords, archive encryption and a fuzzy hash for near duplicate detection. It is organized as one unit per algorithm family (`sgcCrypto_AES`, `sgcCrypto_MLKEM`, `sgcCrypto_TLSH`, and so on), so an application only pulls in the code it actually calls.
Every function is a plain call taking and returning `TBytes`: there are no components to drop on a form and no design time configuration. The implementation of every algorithm listed below was cross checked, byte for byte, against an independent implementation of the same standard (OpenSSL, Python's `hashlib`/`cryptography`, or a reference implementation published by the standard's authors) rather than only against its own output, which is what catches a bug that is internally consistent but wrong.

Enabling the library

The cryptography units belong to the **Cryptography** install package, controlled by the `SGC_PACK_CRYPTO` conditional define in `sgcVer.inc`. With the define off, every `sgcCrypto_*` unit compiles to an empty stub, which keeps the pack out of a build that does not need it. Select the package from the installer, or define `SGC_PACK_CRYPTO` in your project options if you compile from source.
If a project must use only FIPS-approved algorithms, the optional `SGC_CRYPTO_FIPS` define removes every other algorithm from the sgcCrypto units at compile time. It is off by default. See FIPS Mode for what it keeps, what it removes, and why it is not a FIPS 140-3 validation.

The sgcCrypto units are also what lets the rest of the library run without OpenSSL: the `sgcCryptoBackend` runtime switch (or the optional `SGC_NATIVE_CRYPTO` define) and the `iohNativeTLS` IOHandler. See Running without OpenSSL.

What is covered on this page

The table below is the complete inventory: every algorithm the library implements, and which unit it lives in. The pages that follow explain each family one by one, with the exact function signatures and a short Delphi example.

Category	Algorithms	Standard	Unit
Hashing	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512	FIPS 180-4	`sgcCrypto_SHA2`
Hashing	SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, SHAKE256, cSHAKE, KMAC	FIPS 202, SP 800-185	`sgcCrypto_Keccak`
Hashing	BLAKE2b (keyed, up to 512 bit)	RFC 7693	`sgcCrypto_Blake2b`
Hashing	BLAKE2s (keyed, up to 256 bit)	RFC 7693	`sgcCrypto_Blake2s`
Hashing	SipHash-2-4	—	`sgcCrypto_SipHash`
Hashing	RIPEMD-160, HMAC-RIPEMD160	ISO/IEC 10118-3, RFC 2286	`sgcCrypto_Legacy`
Hashing (legacy)	MD4, MD5, DES-ECB, NTLM v1 hash	—	`sgcCrypto_Legacy`
Fuzzy hashing	TLSH digest and distance	—	`sgcCrypto_TLSH`
Message authentication	HMAC-SHA1/256/384/512, HMAC-MD5	FIPS 198-1	`sgcCrypto_HMAC`
Message authentication	AES-CMAC	RFC 4493	`sgcCrypto_CMAC`
Message authentication	AES-GMAC	SP 800-38D	`sgcCrypto_CMAC`
Message authentication	Poly1305	RFC 8439	`sgcCrypto_Poly1305`

Password hashing & KDF	PBKDF2 (SHA-1/256/384/512)	RFC 8018	`sgcCrypto_KDF`
Password hashing & KDF	KDF1, KDF2, ANSI X9.63 KDF	ISO 18033-2, ANSI X9.63	`sgcCrypto_KDF`
Password hashing & KDF	scrypt	RFC 7914	`sgcCrypto_Scrypt`
Password hashing & KDF	Argon2d, Argon2i, Argon2id	RFC 9106	`sgcCrypto_Argon2`
Password hashing & KDF	HKDF (SHA-256/384)	RFC 5869	`sgcCrypto_HKDF`
Symmetric encryption	AES-128/192/256: ECB, CBC, CFB, OFB, CTR, CBC ciphertext stealing, GCM	FIPS 197, SP 800-38A/D	`sgcCrypto_AES`, `sgcCrypto_Modes`
Symmetric encryption	AES-CBC with no padding, and the PKCS#7, zero, ANSI X9.23, ISO 7816-4, ISO 10126-2 and TBC padding schemes	SP 800-38A, ISO/IEC 9797-1	`sgcCrypto_AES`, `sgcCrypto_Modes`
Symmetric encryption	AES-CCM (AEAD)	RFC 3610, SP 800-38C	`sgcCrypto_Modes`
Symmetric encryption	AES Key Wrap, AES Key Wrap with Padding	RFC 3394, RFC 5649	`sgcCrypto_Modes`
Symmetric encryption	ChaCha20, XChaCha20, Salsa20, XSalsa20	RFC 8439	`sgcCrypto_ChaCha`
Symmetric encryption	ChaCha20-Poly1305, XChaCha20-Poly1305 (AEAD)	RFC 8439	`sgcCrypto_Poly1305`
Public key cryptography	RSA key generation, RSA-OAEP, RSA-PSS, PKCS#1 v1.5 sign/verify	RFC 8017	`sgcCrypto_RSA_Keys`, `sgcCrypto_RSA`
Public key cryptography	RSA PKCS#1 v1.5 encryption	RFC 8017	`sgcCrypto_RSA_Keys`
Public key cryptography	Ed25519, Ed448 (signatures)	RFC 8032	`sgcCrypto_Ed25519`, `sgcCrypto_Ed448`
Public key cryptography	X25519, X448 (key agreement)	RFC 7748	`sgcCrypto_X25519`, `sgcCrypto_X448`
Public key cryptography	ECIES sealed box encryption	—	`sgcCrypto_ECIES`
Public key cryptography	ECDSA / ECDH on NIST P-256, P-384, P-521	FIPS 186-5	`sgcCrypto_EC`, `sgcCrypto_ECCurves`
Public key cryptography	ECDSA / ECDH on secp256k1, brainpoolP256r1, brainpoolP384r1, brainpoolP512r1	SEC 2, RFC 5639	`sgcCrypto_ECCurves`
Public key cryptography	ECDSA DER encoded signatures, raw and DER conversion	RFC 3279	`sgcCrypto_ECCurves`, `sgcCrypto_EC`
Public key cryptography	Schnorr signatures on secp256k1	BIP-340	`sgcCrypto_ECCurves`
Public key cryptography	RSA and EC key export and import: PKCS#1, PKCS#8, SEC1, SubjectPublicKeyInfo, DER and PEM	RFC 8017, RFC 5958, RFC 5280	`sgcCrypto_RSA_Keys`, `sgcCrypto_ECCurves`
Post-quantum cryptography	ML-KEM-512/768/1024, hybrid X25519 + ML-KEM-768, X-Wing, X25519MLKEM768, SecP256r1MLKEM768, SecP384r1MLKEM1024	FIPS 203, RFC 9935, RFC 10024, draft-connolly-cfrg-xwing-kem	`sgcCrypto_MLKEM`, `sgcCrypto_MLKEM_Hybrid`

Post-quantum cryptography	ML-DSA-44/65/87	FIPS 204, RFC 9881, RFC 9964	`sgcCrypto_MLDSA`
Post-quantum cryptography	SLH-DSA-SHAKE-128s/128f/192s/192f/256s/256f	FIPS 205, RFC 9909	`sgcCrypto_SLHDSA`
Certificates & PKI	X.509 parsing, chain and CRL verification	RFC 5280	`sgcCrypto_X509`
Certificates & PKI	X.509 self-signed certificate and PKCS#10 CSR generation	RFC 5280, RFC 2986	`sgcCrypto_X509_Gen`
Certificates & PKI	EC signed X.509 certificates and CSRs, URI subject alternative names	RFC 5280, RFC 2986	`sgcCrypto_X509_Gen`
Certificates & PKI	PKCS#12 (.pfx, .p12) reading: leaf certificate, chain and private key. PBES2 with AES, and the legacy Triple DES and RC2 schemes	RFC 7292, RFC 9579	`sgcCrypto_PKCS12`
Certificates & PKI	DER / ASN.1 reader and writer, PEM, Base64	—	`sgcCrypto_ASN1`, `sgcCrypto_DER`
One time passwords	HOTP, TOTP	RFC 4226, RFC 6238	`sgcCrypto_OTP`
Archive encryption	WinZip AE-1 / AE-2	WinZip AE-2	`sgcCrypto_Zip_AE2`
Encoding & utilities	Hex, Base32, Base64url, constant time compare, secure zero, random bytes	RFC 4648	`sgcCrypto_Encoding`, `sgcCrypto_Random`

Explained one by one

1. Hashing & Message Authentication: SHA-2, SHA-3, SHAKE, BLAKE2, SipHash, HMAC, AES-CMAC and Poly1305, the building blocks almost every other algorithm on this page is built from.

2. Password Hashing & Key Derivation: PBKDF2, scrypt, Argon2id and HKDF. Storing a password and deriving a key from an existing secret are different jobs; this page explains why and which function fits which one.

3. Symmetric Encryption: AES in every mode the library supports, AES key wrapping, ChaCha20/Salsa20 and their Poly1305 AEAD constructions, and why an authenticated mode should be the default choice.

4. Public Key Cryptography: RSA, Ed25519/Ed448 signatures, X25519/X448 key agreement, ECIES, ECDSA/ECDH on the NIST, secp256k1 and Brainpool curves, and BIP-340 Schnorr signatures.

5. Post-Quantum Cryptography: ML-KEM, ML-DSA and SLH-DSA, the three schemes standardized by NIST in FIPS 203-205, and the hybrid construction to deploy today.

6. Certificates & PKI: parsing and verifying X.509 certificates you receive, and generating your own self-signed certificates and PKCS#10 requests, signed with either an RSA or an EC key.

7. One Time Passwords: HOTP and TOTP, the codes an authenticator app produces, including the provisioning URI.

8. Fuzzy Hashing (TLSH): a locality sensitive hash for near duplicate detection, unlike every other hash on this page it is not meant for integrity or security.

9. WinZip AE-2 Archive Encryption: the AES encryption used inside .zip archives, and what it does and does not protect.

10. FIPS Mode: a compile time switch that leaves only FIPS-approved algorithms in the library, what it removes and checks, and why it is not a FIPS 140-3 validation.

Demos

The `Demos\18.Cryptography` folder has seven runnable applications, one per area above (hashing, symmetric encryption, passwords, public key, post-quantum, certificates, and one time passwords with WinZip AE-2), each showing the API in use and explaining the reasoning behind the recommended choices.

Hashing & Message Authentication

A cryptographic hash turns any amount of data into a short, fixed size digest. The same input always produces the same digest, a different input (even one changed bit) produces a completely different one, and there is no practical way to find an input that produces a chosen digest. A message authentication code (MAC) adds a secret key to that idea, so the digest also proves the data was not altered by anyone who does not hold the key.

All functions on this page take and return `TBytes` and live in `sgcCrypto_SHA2`, `sgcCrypto_Keccak`, `sgcCrypto_Blake2b`, `sgcCrypto_Blake2s`, `sgcCrypto_SipHash`, `sgcCrypto_HMAC`, `sgcCrypto_CMAC` and `sgcCrypto_Poly1305`.

SHA-2

The hash family behind TLS certificates, code signing and most protocols in production today. Use SHA-256 unless something specific asks for a different size.

SHA-3, SHAKE, cSHAKE and KMAC

SHA-3 is built on a different construction (Keccak) than SHA-2, so a weakness in one family does not carry over to the other, which is why protocols that need long term assurance often specify both. SHAKE128 and SHAKE256 are extendable output functions: instead of a fixed digest size, you ask for as many bytes as you need, and a shorter output is always a prefix of a longer one from the same input. ML-KEM, ML-DSA and SLH-DSA (see Post-Quantum Cryptography) are all built on SHAKE internally. KMAC is the SHA-3 native message authentication code, built on cSHAKE.

BLAKE2b and BLAKE2s

BLAKE2 is faster than SHA-2 and SHA-3 in software while offering the same security margin, and it takes an optional key directly, without the HMAC construction. BLAKE2b targets 64-bit platforms with digests up to 512 bits; BLAKE2s targets 32-bit and embedded platforms with digests up to 256 bits. A shorter digest is **not** a truncation of the full one, the length is mixed into the hash itself, so a program cannot switch sizes and expect matching output.

SipHash

SipHash-2-4 is a fast, keyed hash designed for hash table lookups. Its job is to stop an attacker feeding your application chosen inputs that all land in the same bucket and turn an O(1) lookup into O(n), a denial of service technique known as hash flooding. It takes exactly 16 bytes of key and produces an 8 byte tag; it is not a general purpose MAC and should not be used to authenticate messages.

HMAC

HMAC turns any of the hashes above into a message authentication code. It is the standard choice when the underlying hash does not accept a key directly (SHA-1, SHA-2).

AES-CMAC

CMAC builds a MAC out of a block cipher instead of a hash. Use it where a protocol specifies AES-CMAC by name (it is common in smart card and IoT specifications); otherwise HMAC-SHA256 is the more usual default.

Poly1305

Poly1305 is a one time authenticator: it must never be used twice with the same key. In practice it is not called directly, it is the authentication half of the ChaCha20-Poly1305 and XChaCha20-Poly1305 AEAD ciphers on the Symmetric Encryption page, which derive a fresh one time key for every message.

Comparing a MAC or a password hash

Never compare two tags or digests with a plain = when one of them came from an untrusted source. A byte by byte compare that returns as soon as it finds a mismatch takes measurably less time the earlier the mismatch occurs, which is enough of a signal for an attacker to forge a valid tag one byte at a time over many attempts. Use the constant time compare from `sgcCrypto_Encoding` instead, listed under Encoding & utilities on the Cryptography overview page.

RIPEMD-160

A 160 bit digest from the European RIPE project, in `sgcCrypto_Legacy` but not legacy in the sense the section below means. It is not broken the way MD5 and SHA-1 are, but it is slower than SHA-256 and offers less security margin, so it belongs where a format names it specifically rather than as a default choice. Bitcoin and every chain descended from it use it for addresses, layered over SHA-256. HMAC-RIPEMD160 is RFC 2286.

Legacy hashes

`sgcCrypto_Legacy` provides MD4, MD5 and HMAC-MD5. All three are broken as security primitives: practical collisions have been published for both MD4 and MD5. They exist only for interoperating with older protocols and file formats that specify them by name (for example NTLM authentication). Do not choose them for anything new; use SHA-256 or BLAKE2b instead.

Password Hashing & Key Derivation

These four functions solve two different problems that are easy to confuse.

Storing a password needs to be slow and memory hard, so that stealing the database does not hand over the passwords. Use Argon2id first, scrypt second, and PBKDF2 only where a standard specifically requires it.

Deriving a key from an already random, high entropy secret, for example the output of an X25519 key agreement, needs to be fast: HKDF is the right tool, and pointing a slow password hash at a secret that is already random wastes CPU time for no security benefit.

All functions live in `sgcCrypto_KDF`, `sgcCrypto_Scrypt`, `sgcCrypto_Argon2` and `sgcCrypto_HKDF`.

PBKDF2

Stretches a password by repeating an HMAC many times. It only costs CPU time, so custom hardware (GPUs, FPGAs) attacks it far more cheaply than it costs a legitimate server to check one password. It is the weakest of the three password hashes here, kept because some standards (WPA2, several PKCS formats) still require it by name.

scrypt

`N` is the memory/CPU cost parameter: 16384 with `r`=8 needs roughly 16 MB of RAM per hash, which is what makes a large scale GPU attack expensive rather than merely slow, because the attacker has to pay for memory, not only for compute cores.

Argon2

Argon2 won the 2015 Password Hashing Competition and is the current first choice for storing passwords. **Argon2id** is the variant to use: it mixes the data dependent addressing of Argon2d (strong against GPU cracking) with the data independent first pass of Argon2i (resistant to a side channel attacker who can observe memory access patterns). RFC 9106 recommends at least 19 MiB of memory with 2 iterations for interactive logins, or 64 MiB with 3 iterations where the extra cost is acceptable.

Argon2d and Argon2i are also available through the general entry point for callers that need a specific variant:

HKDF

HKDF splits one good secret into as many independent keys as a protocol needs. The `aInfo` parameter is what keeps them independent: the same secret with a different info string produces an unrelated key, which is exactly what stops a key derived for one purpose (say, encrypting a message) from also being usable for another (say, authenticating one).

The extract and expand steps behind `sgcHKDF_SHA256` are also available separately, for protocols that need to run them at different times:

KDF1, KDF2 and ANSI X9.63

Three counter mode key derivation functions from ISO 18033-2 and IEEE 1363a, also in `sgcCrypto_KDF`. They are the ones RSA-KEM, ECIES and the older ANSI schemes name, which is the only reason to reach for them. HKDF above remains the better choice for anything new, because it separates extraction from expansion instead of hashing the secret straight into the output.

KDF1 and KDF2 differ in exactly one detail, and it is a real interoperability trap: KDF1 starts its counter at 0, KDF2 at 1. The outputs are completely different, each function looks correct in isolation, and a specification does not always spell out which one it means, so check the spec rather than the name. X9.63 is KDF2 with a shared-info string appended after the counter, which is the form ECIES and the older ECDH TLS suites specify.

Symmetric Encryption

Symmetric encryption uses the same key to encrypt and decrypt. The one idea that matters most on this page is the difference between a plain cipher and an AEAD (Authenticated Encryption with Associated Data) cipher. AES-CBC or AES-CTR alone give confidentiality and nothing else: an attacker who cannot read the plaintext can still flip bits in the ciphertext, and the recipient will decrypt the tampered result without any indication that anything is wrong. AES-GCM and ChaCha20-Poly1305 authenticate as well as encrypt, so tampering is detected and decryption fails outright. Prefer an AEAD cipher unless something specific forces a plain mode.

All functions live in `sgcCrypto_AES`, `sgcCrypto_Modes`, `sgcCrypto_ChaCha` and `sgcCrypto_Poly1305`. **Never reuse an IV or a nonce with the same key**: for CTR and the AEAD ciphers, reuse does not just weaken confidentiality, for GCM specifically it breaks the authentication guarantee outright. Generate a fresh one with `sgcRandomBytes` for every message.

AES-GCM (recommended)

The AEAD mode most protocols default to. The nonce is 12 bytes; generate it at random per message. `aAAD` is authenticated but not encrypted, which is how a ciphertext gets bound to its context, for example a message id or a protocol version, without hiding that context from anyone who can see the ciphertext.

ChaCha20-Poly1305 and XChaCha20-Poly1305

The RFC 8439 AEAD pair. Where AES-GCM leans on AES hardware acceleration, ChaCha20-Poly1305 runs at constant speed in pure software, which is why TLS and WireGuard default to it on mobile and embedded devices without AES-NI. The nonce is 12 bytes, same restriction as GCM. XChaCha20-Poly1305 extends the nonce to 24 bytes, large enough to pick at random for the lifetime of a key without tracking a counter, which matters when many independent writers share one key and cannot coordinate a counter between them.

AES-CCM

Counter with CBC-MAC, RFC 3610 and NIST SP 800-38C. This is the AEAD that constrained and IoT stacks favour, because it needs only the block cipher itself and no GHASH table: Zigbee, 802.15.4, Bluetooth and the TLS CCM suites all specify it. Where there is room, GCM remains the better pick, since CCM makes two passes over the data and cannot be pipelined.

The nonce is 7 to 13 bytes and its length is not a free choice: 15 minus the nonce length is the size of the length field, so it fixes the maximum message size, and a 13 byte nonce caps the plaintext at 65535 bytes. **A nonce must never repeat under one key**, exactly as with GCM. `aTagLen` is 4, 6, 8, 10, 12, 14 or 16 bytes.

AES-GMAC

GMAC is exactly AES-GCM run with an empty plaintext and the whole message passed as additional data, which is how the standard defines it. It therefore inherits every one of GCM's nonce rules: `aIV` is normally 12 bytes and must never repeat under one key. Reusing an IV under GMAC does not merely weaken it, it leaks the hash subkey and lets an attacker forge tags outright, so the failure is total rather than gradual. AES-CMAC on the Hashing & Message Authentication page needs no IV at all and is the safer pick unless GCM is already in the protocol. The tag is 16 bytes.

AES block modes

These give confidentiality only, with no authentication: pair with a MAC from Hashing & Message Authentication if the plaintext needs integrity, or use AES-GCM above instead.

`sgcAES_CTR` and `sgcAES_OFB` are their own inverse: call the same function again with the same key and IV to decrypt. ECB is included only for interoperating with old formats that specify it; identical plaintext blocks produce identical ciphertext blocks, which leaks the structure of the data, so it should not be chosen for anything new.

CBC is also available with no padding at all, the mode Java writes as `AES/CBC/NoPadding`. The input length must be an exact multiple of 16 and an exception is raised when it is not, rather than the call quietly padding and producing a ciphertext the other side cannot read. It is only safe where the protocol already guarantees block aligned data.

Ciphertext stealing avoids the padding overhead of CBC for data that is not a multiple of the block size:

Padding schemes

A block mode needs the plaintext to be a multiple of 16 bytes, and the padding scheme is how it gets there. PKCS#7 is the only right choice for something new; the other six exist to read and write data from a system that already chose otherwise. `padPKCS5` is not in the list because for a 16 byte block it is PKCS#7, the two names differ only in the block sizes they were defined for.

`sgcAES_CBC_Encrypt` and `sgcAES_ECB_Encrypt` further up are the PKCS#7 shorthand for these. `sgcPad_Remove` returns `False` on a malformed pad rather than raising, and takes the same path for every kind of failure, because distinguishing them is what a padding oracle needs. Take particular care with `padZero`: it cannot distinguish a padding byte from a trailing plaintext zero, so it strips every trailing zero, and data that can legitimately end in one comes back truncated.

AES hardware acceleration (AES-NI)

On Windows Win32 and Win64 builds made with Delphi, sgcCrypto_AES checks the CPU at run time and, when it reports AES-NI, PCLMULQDQ and SSSE3, runs single block encryption, AES-CTR, AES-GCM (the counter mode

and GHASH) and AES-CBC encryption on those instructions, which are constant time by design. Without them, and on Linux64, macOS, iOS and Android, the constant time software core is used as before. CBC decryption always uses the software core. The native TLS engine benefits from it automatically.
`sgcAES_HardwareAvailable` returns True when this build carries the hardware path and the CPU supports it. The global variable `sgcAES_UseHardware`, True by default, set to False forces the software core, for example to compare both. Set it once at start up.

```
var sgcAES_UseHardware: Boolean = True;
function sgcAES_HardwareAvailable: Boolean;

if sgcAES_HardwareAvailable then
  Memo1.Lines.Add('AES-NI in use');
```

AES Key Wrap
Key wrapping protects a **key**, not a message: it needs no IV, it is deterministic, and it authenticates the wrapped key. This is the standard way to store a data encryption key under a master key, for example an HSM or KMS master key wrapping a per-file key. RFC 3394 requires the key being wrapped to be a multiple of 8 bytes; RFC 5649 adds padding for keys of any length.
ChaCha20 and Salsa20 (raw stream ciphers)
The stream ciphers behind the AEAD constructions above, provided directly for protocols that specify the raw cipher. Neither authenticates: almost always the Poly1305 AEAD versions further up this page are what you want instead. Salsa20 is the predecessor of ChaCha20, kept for compatibility with NaCl era formats; choose ChaCha20 for anything new.
Legacy: DES and NTLM
`sgcCrypto_Legacy` also provides single DES in ECB mode and the DES based NTLM v1 key derivation. DES has a 56 bit effective key, brute forceable on modern hardware in hours; it exists only to interoperate with protocols that specify it by name (NTLM authentication is the usual reason). Do not choose it for anything new; use AES instead.

Public Key Cryptography

Public key (asymmetric) cryptography uses a key pair instead of a shared secret: whatever the private key signs, the matching public key verifies; whatever is sealed to a public key, only the matching private key opens. The public key can be handed to anyone, including an adversary, without weakening anything.

RSA

Key generation searches for large primes and is genuinely slow, several seconds for a 2048 bit key; never run it on a UI thread. Use OAEP for encryption and PSS for signatures, both include a randomized element that PKCS#1 v1.5 lacks. PKCS#1 v1.5 sign/verify is provided in `sgcCrypto_RSA` for interoperating with formats that specify it (JWS `RS256` is the common case); prefer PSS for anything new.

PKCS#1 v1.5 encryption is provided as well, the mode Java names `RSA/NONE/PKCS1Padding`. It is here for interoperating with formats that mandate it and for nothing else: v1.5 padding is exactly what the Bleichenbacher attack targets, so OAEP stays the right choice for anything new. The plaintext is limited to the modulus size minus 11 bytes, 245 bytes with a 2048 bit key.

`sgcRSA_PKCS1_Decrypt` returns `False` for every kind of decoding failure and takes the same path through each of them, because telling them apart is precisely what a padding oracle needs.

OAEP with a separate MGF1 hash

OAEP uses a hash in two places, over the label and inside the MGF1 mask generation function, and the two do not have to be the same. That is where Java bites. SunJCE reads `OAEPWithSHA256AndMGF1Padding` as SHA-256 for the digest with MGF1 still on SHA-1, while Bouncy Castle reads the same string as SHA-256 for both, so a ciphertext one of them produces is undecryptable by the other. The five parameter overloads pin the two hashes independently, which is what talking to a Java peer can require. The four parameter form is unchanged and passes `aHash` for both.

A hash mismatch is indistinguishable from a corrupt ciphertext, so a wrong `aMGFHash` comes back as a plain `False` with nothing to say which of the two is wrong. Both peers have to agree on it in advance.

Keys export to PKCS#1 and SubjectPublicKeyInfo DER, and to PEM:

PKCS#8 is the other private key container, the one written as `BEGIN PRIVATE KEY`, and it is what Java and .NET expect; the pair above emits the older PKCS#1 `BEGIN RSA PRIVATE KEY` instead. Neither container is encrypted, so what comes back is bare key material and belongs nowhere a log or a backup can reach.

Importing an RSA key

Until these existed a `TsgcRSAPrivateKey` could only come out of `sgcRSA_GenerateKey`, which meant a key saved to disk could not be used at all. The import routines read both private key containers, PKCS#1 `RSAPrivateKey` and PKCS#8 `PrivateKeyInfo`, and both public key encodings, PKCS#1 `RSAPublicKey` and X.509 SubjectPublicKeyInfo, as DER or as PEM. An encrypted PKCS#8, the `BEGIN ENCRYPTED PRIVATE KEY` form, is not supported and returns `False`. The CRT parameters are recomputed when the file omits them, so an imported key still runs at full speed.

A certificate is not a public key, so hand `sgcRSA_ImportPublicKeyDER` the SubjectPublicKeyInfo pulled out of the certificate, not the certificate itself. `sgcRSA_CheckPrivateKey` verifies the internal consistency of anything that came from elsewhere, that n = p*q, that e*d is congruent to 1, and that the CRT parameters match.

Ed25519 and Ed448 (signatures)

EdDSA nonces are derived deterministically from the private key and the message, so there is no random number generator involved in signing and no possibility of a repeated nonce leaking the key, unlike ECDSA (see below). `sgcCrypto_Ed25519` is the verify/sign primitive used by the WebAuthn EdDSA credential type elsewhere in sgcWebSockets; it expects the caller to already know the 32-byte public key (WebAuthn hands it to you from the credential) and does not expose a separate key generation or public-key-derivation call. `sgcCrypto_Ed448` is the general purpose signer, with roughly 224 bits of security against 128 for Ed25519, at the cost of a larger key and signature and slower signing.

X25519 and X448 (key agreement)

Diffie-Hellman on Montgomery curves: both sides generate a key pair, exchange public keys, and each computes the same shared secret from its own private key and the other side's public key. Always run the raw output through a KDF (see Password Hashing & Key Derivation, HKDF) before using it to encrypt anything; the raw ECDH output is not itself a suitable key. X448 gives roughly 224 bits of security against 128 for X25519, at the cost of a larger key and about three times the computation.

`SharedSecret` returns `False` for a low order peer key, the small set of public keys that force the shared secret to all zeros regardless of your own private key. Always check the result rather than trusting the output unconditionally.

ECIES (sealed box encryption)

Combines X25519, HKDF-SHA256 and XChaCha20-Poly1305 into a single call that encrypts a message to a recipient's public key, with no prior shared secret needed. `aContext` binds the sealed message to a purpose or protocol step, opening under a different context fails, which stops a sealed message from one exchange being replayed into another.

ECDSA / ECDH on named curves

Two separate units cover the short Weierstrass curves, split by which curve family they implement:

`sgcCrypto_EC` covers the NIST curves (P-256, P-384, P-521) and is oriented around PEM keys and JWS signatures, the shape JWT libraries expect (`ES256` and similar).

`sgcCrypto_ECCurves` covers secp256k1 (the Bitcoin and Ethereum curve) and the brainpoolP256r1 / P384r1 / P512r1 curves of RFC 5639, working directly on raw key and signature bytes rather than PEM.

`TsgcECCurve` in `sgcCrypto_ECCurves` now names seven curves rather than four: `eccSecp256k1`, the three Brainpool curves, and `eccP256`, `eccP384` and `eccP521` for NIST P-256, P-384 and P-521. The NIST three used to be reachable only through the PEM and JWS API of `sgcCrypto_EC`, so an application holding a P-256 key as raw bytes had to write PEM first. The raw key generate, sign, verify, ECDH and compress calls below now cover all seven. Reach for `sgcCrypto_EC` when the job is JWS, and for `sgcCrypto_ECCurves` when the key material is already bytes.

Both units declare a type named `TsgcECCurve` and a routine named `sgcEC_GenerateKeyPair`, with different meanings in each unit. Do not add both units to the same `uses` clause and reference either name unqualified; qualify with the unit name (`sgcCrypto_ECCurves.sgcEC_GenerateKeyPair`) if a single unit genuinely needs both.

ECDSA signatures here use RFC 6979 deterministic nonces: the nonce is derived from the private key and the message rather than drawn from a random number generator, which removes the single most common way real deployments have lost an ECDSA private key, a repeated or partially predictable nonce. Signatures carry the low-S normalization that Bitcoin (BIP 62) and Ethereum require; verification accepts either form.

DER encoded ECDSA signatures

One signature, two encodings, and picking the wrong one is among the most common integration bugs in this area. The raw pair R || S, each half the curve field size, is what JOSE and WebAuthn carry. A DER `SEQUENCE { r, s }` is what X.509, CMS and TLS carry. `sgcECDSA_SignHash` above produces the raw form, the calls below produce and consume the DER form, and the two converters translate between them without re-signing anything.

`aHashKind` in the raw key form only picks the HMAC behind the RFC 6979 nonce, it hashes nothing for you, so pass `sgcEC_DefaultHash(aCurve)` when there is no reason to choose otherwise. `sgcECDSA_RawToDER` returns an empty array when the pair is not exactly two field size halves, and `sgcECDSA_DERToRaw` returns `False` on malformed DER or on a value that does not fit the curve, so both are safe to point at untrusted input.

Exporting and importing EC keys

Three containers, and the PEM label is what tells them apart: a SEC1 `ECPrivateKey` is `BEGIN EC PRIVATE KEY`, a PKCS#8 `PrivateKeyInfo` is `BEGIN PRIVATE KEY`, and an X.509 SubjectPublicKeyInfo is `BEGIN PUBLIC KEY`. Java and .NET want PKCS#8, OpenSSL and most command line tooling read either. Neither private form is encrypted, so the result is as secret as the scalar itself.

The import routines read the curve out of the file, which is why `aCurve` comes back as an `out` parameter instead of being passed in. A curve the unit does not implement is a negotiation failure rather than a bug, so they return `False` and never raise, which makes them safe on untrusted input. The public point is recomputed from the scalar when the file leaves it out, and validated against the curve before it is handed back. The export routines raise instead, because key material that does not match the curve is a caller bug and not input.

The dotted named curve OID that goes into each of those containers is available directly, and is what the export and import routines use internally:

Always validate a peer's public key

Before using any public key you did not generate yourself, whether for ECDH, ECDSA verification or ECIES, validate that it is actually a point on the curve with `sgcEC_ValidatePublicKey`. A point that is not on the curve can leak bits of your own private key through the arithmetic used to process it.

Schnorr signatures (BIP-340)

secp256k1 only, and this is what Taproot, Nostr and Lightning use. It is not a drop-in replacement for the ECDSA above, because two things differ. Public keys are x-only, 32 bytes with the y coordinate implied to be even, so a BIP-340 key is not interchangeable with the 33 byte compressed or the 65 byte uncompressed forms the rest of this page produces. And the challenge hashes the public key together with the nonce and the message, which binds the signature to the key in a way an ECDSA signature is not.

`aAuxRand` is 32 bytes of auxiliary randomness mixed into the nonce derivation. It is defence in depth against fault attacks, not the nonce itself: the scheme stays secure with it all zeroes, and an empty array is treated as 32 zero bytes so that the BIP-340 test vectors reproduce exactly. Real use should pass fresh randomness. `sgcSchnorr_TaggedHash` is exposed because BIP-341 and BIP-342 build further tags on the same construction.

Post-Quantum Cryptography

A sufficiently large quantum computer would break RSA, X25519/X448 and ECDSA/ECDH on every curve on the Public Key Cryptography page: none of that math resists Shor's algorithm. It does not need to exist today to be a problem today, an adversary can record encrypted traffic now and decrypt it once such a machine exists, a strategy usually called "harvest now, decrypt later". Anything that must stay confidential for years, not days, should already be moving to post-quantum key exchange.
NIST standardized three schemes, all implemented here, and all built on nothing but SHAKE (see Hashing & Message Authentication), so their security rests only on the hash being sound: no lattice or number theory assumption to break.

Scheme	Purpose	Standard	Unit
ML-KEM	Key encapsulation, replaces Diffie-Hellman	FIPS 203	`sgcCrypto_MLKEM`
ML-DSA	Signatures, lattice based, the default choice	FIPS 204	`sgcCrypto_MLDSA`
SLH-DSA	Signatures, hash based only, conservative fallback	FIPS 205	`sgcCrypto_SLHDSA`

ML-KEM and the hybrid construction (recommended)
ML-KEM replaces a Diffie-Hellman exchange: the sender encapsulates a shared secret to the recipient's public key, the recipient decapsulates it with the private key, and both end up with the same 32 bytes. It comes in three parameter sets, larger meaning a bigger security margin and bigger keys.
FIPS 203 decapsulation never fails.
A corrupt or malicious ciphertext does not raise an error, it silently returns a different secret, derived from an implicit rejection value baked into the private key. Returning an error instead would hand an attacker a decryption oracle to probe the private key with; the standard avoids that by design. Whatever the derived secret protects has to detect the mismatch on its own, for example an AEAD tag failing to verify downstream.
What to actually deploy today is the hybrid construction: X25519 combined with ML-KEM-768, so the shared secret depends on both. It stays safe if ML-KEM is ever broken by a cryptanalytic advance, and it stays safe if a quantum computer breaks X25519; an attacker needs both to fail at once.
X-Wing and the TLS 1.3 hybrid groups
`sgcCrypto_MLKEM_Hybrid` holds three hybrid families. The `sgcHybrid_` functions above are the original sgcCrypto format and are kept for compatibility. `X-Wing` is the standardised general purpose hybrid KEM, X25519 combined with ML-KEM-768, and it is the one to use in new code outside TLS. The third family builds the TLS 1.3 hybrid key shares and nothing else.
X-Wing (draft-connolly-cfrg-xwing-kem-10) stores one secret only: the private key is a 32-byte seed, and every operation expands it into the ML-KEM-768 and X25519 material it needs. The public key is 1216 bytes, the ciphertext 1120 bytes and the shared secret 32 bytes. Encapsulation runs the FIPS 203 encapsulation key check and raises when the public key fails it.
The TLS 1.3 hybrid key shares of RFC 10024 are `X25519MLKEM768`, `SecP256r1MLKEM768` and `SecP384r1MLKEM1024`, IANA codepoints 0x11EC, 0x11EB and 0x11ED. These are not KEMs you can use on their own. The shared secret is a plain concatenation of the two halves, which is safe only because TLS feeds it into its key schedule together with the handshake transcript, so use them to build a TLS key share and nothing else. The Native TLS engine offers `X25519MLKEM768` first by default, so a client or a server using it already negotiates a post-quantum key exchange with no extra code.
The hybrid groups exist only in TLS 1.3. The native engine also speaks TLS 1.2, and a connection that falls back to it because the peer cannot do TLS 1.3 uses a classic group, X25519, secp256r1 or secp384r1, with no post-quantum protection. Set `TLSOptions.Version` on a client or `SSLOptions.Version` on a server to `tls1_3`, and list only hybrid groups in `NativeTLS_Options.Groups`, when the key exchange must always be post-quantum.
The Post-Quantum TLS demo, `Demos\18.Cryptography\08.PostQuantum_TLS`, has a TLS version selector with the choices TLS 1.3 and TLS 1.2, TLS 1.3 only and TLS 1.2 only, and a cipher suite selector, and its status line shows the negotiated version, group and cipher suite. The server demo can sign with an ML-DSA-44 certificate, which serves TLS 1.3 clients only, because TLS 1.2 has no ML-DSA signature scheme.
The client keeps `aPrivateState` secret, passes it unchanged to `sgcTLSHybrid_ClientSharedSecret` and wipes it afterwards; its layout is not part of the API. The two Boolean functions return False, with every output empty, on any invalid peer input: a key share of the wrong length, an ML-KEM encapsulation key that fails the FIPS 203 check, a point that is not on the curve, or an all-zero X25519 result. RFC 10024 requires the connection to be aborted with an `illegal_parameter` alert in each of those cases.
Everything built on X25519, which means the `sgcHybrid_` functions, X-Wing and the `X25519MLKEM768` group, is compiled out of an `SGC_CRYPTO_FIPS` build, see FIPS Mode. The two SecP groups stay, because P-256, P-384

and ML-KEM are all approved algorithms. Call `sgcTLSHybrid_Available` before offering or accepting a group rather than assuming it is there.

ML-DSA (recommended for signatures)

Signing draws a masking vector and tries to build a signature, restarting when the attempt would leak information about the key; the expected number of attempts is small but not fixed, so signing takes a variable amount of time. That is inherent to the scheme and is not a side channel on the key. Signatures are large: 2420 bytes for ML-DSA-44 against 64 for Ed25519, and public keys 1312 bytes against 32, so budget for the difference before adopting it in a size sensitive protocol.

SLH-DSA (conservative fallback)

SLH-DSA rests on nothing but the hash function: no lattice problem to break, ever, at any point in the future. That guarantee is expensive. A signature runs from 7856 bytes (the smallest, `SLH-DSA-SHAKE-128s`) to 49856 bytes (the largest, `SLH-DSA-SHAKE-256f`), and signing an `s` (small) parameter set walks hundreds of thousands of hash operations and can take tens of seconds; the `f` (fast) sets sign in around one to seven seconds at roughly twice the signature size. Verification is quick for every parameter set, which suits a sign-rarely, verify-often use case such as firmware or code signing better than an interactive protocol.

Key formats: SubjectPublicKeyInfo, PKCS#8 and PEM

The raw byte strings the FIPS algorithms consume are not a storage format, and no other implementation will read them. To write a key to disk, put it in a certificate or hand it to another library, export it in the standard containers: an X.509 SubjectPublicKeyInfo for a public key, a PKCS#8 OneAsymmetricKey for a private key, and PEM armour (RFC 7468) around either, labelled `PUBLIC KEY` and `PRIVATE KEY`. The three schemes expose the same eight functions, each following the RFC that defines its encoding.

Scheme	Container RFC	DER functions	PEM functions
ML-DSA	RFC 9881	`sgcMLDSA_ExportSubjectPublicKeyInfo`, `sgcMLDSA_ImportSubjectPublicKeyInfo`, `sgcMLDSA_ExportPrivateKeyInfo`, `sgcMLDSA_ImportPrivateKeyInfo`	`sgcMLDSA_ExportPublicKeyPEM`, `sgcMLDSA_ImportPublicKeyPEM`, `sgcMLDSA_ExportPrivateKeyPEM`, `sgcMLDSA_ImportPrivateKeyPEM`
ML-KEM	RFC 9935	`sgcMLKEM_ExportSubjectPublicKeyInfo`, `sgcMLKEM_ImportSubjectPublicKeyInfo`, `sgcMLKEM_ExportPrivateKeyInfo`, `sgcMLKEM_ImportPrivateKeyInfo`	`sgcMLKEM_ExportPublicKeyPEM`, `sgcMLKEM_ImportPublicKeyPEM`, `sgcMLKEM_ExportPrivateKeyPEM`, `sgcMLKEM_ImportPrivateKeyPEM`
SLH-DSA	RFC 9909	`sgcSLHDSA_ExportSubjectPublicKeyInfo`, `sgcSLHDSA_ImportSubjectPublicKeyInfo`, `sgcSLHDSA_ExportPrivateKeyInfo`, `sgcSLHDSA_ImportPrivateKeyInfo`	`sgcSLHDSA_ExportPublicKeyPEM`, `sgcSLHDSA_ImportPublicKeyPEM`, `sgcSLHDSA_ExportPrivateKeyPEM`, `sgcSLHDSA_ImportPrivateKeyPEM`

The algorithm identifier carries the NIST object identifier with the parameters absent, so the parameter set travels with the key and import returns it. `sgcMLDSA_OID`, `sgcMLKEM_OID` and `sgcSLHDSA_OID` give the dotted string, and the matching `ParamsFromOID` functions map one back.

Three private key forms for ML-DSA and ML-KEM.

The export functions take an `aFormat` parameter of type `TsgcPQCPrivateKeyFormat`, declared in `sgcCrypto_DER`. `pqkfSeed`, the default, writes the seed alone, 32 bytes for ML-DSA and the 64-byte `d || z` for ML-KEM; this is the form both RFCs recommend and it is by far the smallest. `pqkfExpanded` writes the expanded FIPS private key. `pqkfBoth` writes both and raises when the expanded key is not the expansion of the seed, which is what the RFCs call a malformed key.

A seed cannot be recovered from an expanded key afterwards, so ask for it at generation time with `sgcMLDSA_GenerateKeyPairAndSeed` or `sgcMLKEM_GenerateKeyPairAndSeed` when you intend to store the recommended form. SLH-DSA has a single private key form, `SK.seed || SK.prf || PK.seed || PK.root`, so its export takes no format parameter.

Import validates, it does not trust.

Every import function returns False rather than raising, and it returns False on malformed DER, on trailing data, on an unknown object identifier, on parameters that are present when they must be absent, and on a key whose length does not match the parameter set. Beyond the structure, an ML-KEM public key must pass the FIPS 203 modulus check, an expanded ML-KEM private key must pass the hash check and the pairwise consistency test, and an expanded ML-DSA private key is rebuilt into its public key so that the stored `t0` and `tr` must both agree with it. An embedded public key that does not match the private key is refused. When a private key holds a seed, all three outputs come back filled, so a caller never has to expand it separately.

`sgcSLHDSA_ImportPrivateKeyInfo` and `sgcSLHDSA_ImportPrivateKeyPEM` take an extra `aVerify` parameter, True by default, that recomputes `PK.root` from the private seed and refuses a key whose stored root differs. That rebuild costs as much as a key generation, which is seconds for the `s` parameter sets, so pass False only for keys your own application generated or already trusts. The object identifier, the lengths and the structure are still checked when it is False.

These are the containers the rest of the library reads. A PKCS#8 ML-DSA private key is what the JWT client signs with, a SubjectPublicKeyInfo is what the JWT server validates with, and both are what Certificates & PKI puts into a certificate or a certificate signing request.

Choosing a parameter set

Every size call (`sgcMLKEM_PublicKeySize`, `sgcMLDSA_SignatureSize`, `sgcSLHDSA_SignatureSize` and their `PrivateKeySize` counterparts) takes the same parameter enum as the operation itself, so a protocol can size its buffers before doing any cryptography:

Parameter set	Public key	Private key	Ciphertext / Signature
ML-KEM-512	800	1632	768
ML-KEM-768	1184	2400	1088
ML-KEM-1024	1568	3168	1568
ML-DSA-44	1312	2560	2420
ML-DSA-65	1952	4032	3309
ML-DSA-87	2592	4896	4627
SLH-DSA-SHAKE-128s	32	64	7856
SLH-DSA-SHAKE-128f	32	64	17088
SLH-DSA-SHAKE-192s	48	96	16224
SLH-DSA-SHAKE-192f	48	96	35664
SLH-DSA-SHAKE-256s	64	128	29792
SLH-DSA-SHAKE-256f	64	128	49856

All sizes are in bytes. The ML-KEM shared secret is always 32 bytes regardless of parameter set (`sgcMLKEM_SharedSecretSize`).

Certificates & PKI

Two units cover opposite directions of the same X.509 format: `sgcCrypto_X509` reads and verifies a certificate someone else gave you, `sgcCrypto_X509_Gen` writes a certificate or request of your own. A third unit, `sgcCrypto_ASN1`, provides the DER/PEM plumbing both of them sit on.

Reading and verifying a certificate

`sgcCrypto_X509` parses a DER encoded certificate into a plain record, and checks a chain or a revocation list against it. Use this side of the API when your application receives a certificate over the wire and has to decide whether to trust it.

`sgcX509_VerifyChain` takes the chain and the trust anchors as DER blobs, not as parsed records, and a Unix timestamp to test the validity window against. Element 0 of the chain is the leaf and each certificate must be signed by the next one; the top must be signed by, or be equal to, one of the trust roots. The path is rejected when a certificate is outside its validity window at `aNowUnix`, when an issuer is not a CA, when it breaks `basicConstraints pathLenConstraint` (self-issued certificates do not count against the limit, RFC 5280 section 6.1.4), or when it breaks a `nameConstraints` extension on any certificate in the path, the trust anchor included. The constraint choices checked are `dNSName`, `iPAddress`, `rfc822Name`, `uniformResourceIdentifier` and `directoryName`; the DNS subtrees also apply to a leaf common name that reads as a host name when the leaf carries no DNS name of its own, and a critical `nameConstraints` naming any other choice rejects the path. Certificate policies are not checked.

`TsgcX509Cert` exposes the fields a caller actually needs after parsing: `Version`, `SerialNumber`, the raw `Issuer`/`Subject` RDN sequences, `NotBeforeUnix`/`NotAfterUnix`, the public key (whichever `PubKeyType` says it is, `x509pkRSA`, `x509pkEC`, `x509pkEd25519`, `x509pkMLDSA`, `x509pkSLHDSA` or `x509pkMLKEM`) and `IsCA`. A post-quantum key arrives in `PQCPublicKey`, and the parameter set its object identifier names is in whichever of `MLDSAParams`, `SLHDSAParams` or `MLKEMParams` matches the type. `TsgcX509CSR` mirrors the same fields for a PKCS#10 request.

Generating a certificate or a request

`sgcCrypto_X509_Gen` builds a self-signed certificate, or a PKCS#10 certificate signing request to send to a CA, from an RSA key pair (see Public Key Cryptography). Both take the same options record.

Build the validity window from UTC.

Certificate validity is encoded as UTCTime with a `Z` suffix, meaning Greenwich time, so filling `NotBefore` and `NotAfter` from a local `Now` dates the certificate wrong by your offset from UTC. East of Greenwich a freshly issued certificate is then not yet valid until that offset elapses, west of it the certificate expires early. `sgcX509_UTCNow` returns the right value, and `sgcX509_DefaultOptions` already uses it with a five minute backdate for clock skew, the same margin real CAs apply.

Fill in the subject alternative names.

Modern browsers and TLS libraries ignore the common name entirely and look only at the `DNSNames` extension; a certificate with a perfect common name and no DNS name in `SubjectAltName` will be rejected however correct the rest of it is.

Alongside `DNSNames`, `IPAddresses` and `Emails`, `TsgcX509Options` carries a `URIs` array, for subject alternative names of type `uniformResourceIdentifier`. A service identity that is a URI rather than a host name goes here, the concrete case being a SPIFFE ID, which a workload mesh matches literally and which no DNS name can express. The value goes in verbatim as IA5String content, so pass a full URI.

Set `IsCA := True` and a non-negative `PathLength` to produce a CA certificate rather than a leaf certificate; leave `PathLength` at -1 to omit the constraint on a leaf. A CSR carries the same requested extensions in its `extensionRequest` attribute, which is where a CA looks for them when issuing the final certificate.

IPv6 subject alternative names.

`IPAddresses` takes IPv4 in dotted form and IPv6 in any of the text forms of RFC 4291: the full eight group form, a form compressed with `::`, and a form with an embedded IPv4 address in the last 32 bits, in any letter case. Do not add brackets and do not add a zone index. An IPv4 entry is written as 4 octets and an IPv6 entry as 16, which is what RFC 5280 requires and what a TLS client compares the peer address against.

Name constraints on a CA certificate.

A CA certificate can limit what the certificates below it are allowed to name, RFC 5280 section 4.2.1.10. Fill any of `PermittedDNSNames`, `ExcludedDNSNames`, `PermittedIPRanges` or `ExcludedIPRanges` and the extension is written, marked critical. A DNS entry is a subtree, so `example.com` covers that name and every name below it. An IP entry is an address with a prefix length. `sgcX509_VerifyChain` enforces what you write here.

Issuing from a CA

A self-signed certificate asserts nothing a third party can check. `sgcX509_CreateSigned` issues a certificate from a CA key and the CA's own certificate, and `sgcX509_CreateSignedFromCSR` does the same starting from a PKCS#10 request.

The issuer name is copied byte for byte from the subject of the issuer certificate, so the chain links exactly, and the `authorityKeyIdentifier` is that certificate's `subjectKeyIdentifier`. `TsgcX509Options.Issuer` is never read by any builder. Both functions raise before signing anything when the issuer certificate is malformed, is not a CA, or carries a key usage without `keyCertSign`, when the key you passed is not the key of that certificate, and when the subject key is malformed or of an algorithm the unit does not know.
When issuing from a request, the signature on the request is verified first and a request that fails raises. The subject name and the public key come from the request, so `aOptions.Subject` is ignored, and the extensions come from `aOptions` alone. What the request asked for is not copied: the CA decides what the certificate asserts.

Signing with an EC key

`sgcX509_CreateSelfSigned` and `sgcX509_CreateCSR` sign with RSA. The `Ex` pair takes a `TsgcX509SignKey` instead, which wraps either kind of key, so the same options record produces an RSA or an EC certificate depending on what you hand it. Build the wrapper with `sgcX509_RSAKey` or `sgcX509_ECKey` rather than filling the record by hand, the unused half is left undefined on purpose. The two original functions are unchanged and now simply wrap `sgcX509_RSAKey` internally.
A P-256 certificate is roughly a third the size of a 2048 bit RSA one and verifies far faster, which is why modern deployments default to EC. `TsgcX509Options.Hash` still only names the SHA; the signature algorithm follows the key kind, so `rhSHA256` with an EC key means `ecdsa-with-SHA256`, and with an RSA key it means `sha256WithRSAEncryption`.
`sgcX509_ECKey` derives the public point from the scalar, so you never have to pass it. The curve comes from `sgcCrypto_ECCurves` and can be any of the seven it implements, though a public CA will only accept P-256 or P-384.

Post-quantum certificates

The same `Ex` builders sign with a post-quantum key. `sgcX509_MLDSAKey` wraps an ML-DSA key pair and `sgcX509_SLHDSAKey` an SLH-DSA private key, and from there nothing else changes: the same options record produces a self-signed certificate or a certificate signing request signed with ML-DSA or SLH-DSA. See Post-Quantum Cryptography for the schemes themselves and for the key containers the certificate carries.
Both schemes sign in the pure mode with an empty context string, and the signature algorithm is the key's own object identifier with the parameters absent, following RFC 9881 for ML-DSA and RFC 9909 for SLH-DSA. `TsgcX509Options.Hash` is ignored for these keys, since neither hashes the body separately. `sgcX509_MLDSAKey` checks both key lengths and makes the pair sign and verify a fixed message, so a private key handed the wrong public key raises there instead of producing a certificate nothing can verify.

Key usage is restricted.

Only `kuDigitalSignature`, `kuNonRepudiation`, `kuKeyCertSign` and `kuCRLSign` are accepted with an ML-DSA or SLH-DSA key, and the rule is enforced before anything is signed. `sgcX509_DefaultOptions` asks for `kuKeyEncipherment`, so it raises with these keys until you change `KeyUsage`. An empty `KeyUsage` leaves the extension out, which both RFCs allow.

Certifying an ML-KEM key.

An ML-KEM key encapsulates, it never signs, so it can be the subject of a certificate but can never sign a request to ask for one. Pass its SubjectPublicKeyInfo to `sgcX509_CreateSigned` after proving possession by other means, which is what RFC 9935 leaves to the application. `kuKeyEncipherment` is the only key usage flag allowed on such a certificate. For the same reason `sgcX509_CSR_Verify` can never pass on an ML-KEM request, and `sgcX509_VerifySignedBy` never verifies anything with an ML-KEM issuer key.

Reading one back.

`sgcX509_Parse` and `sgcX509_CSR_Parse` treat a post-quantum certificate as malformed when its SubjectPublicKeyInfo or its ML-DSA or SLH-DSA signature algorithm carries parameters, and when the key length does not match the parameter set the object identifier names. An ML-KEM key must additionally pass the FIPS 203 modulus check. `sgcX509_VerifySignedBy` verifies an ML-DSA or SLH-DSA signature in the pure mode with an empty context, and returns False when the parameter set named by the signature algorithm is not the issuer key's own.

DER, ASN.1 and PEM infrastructure

`sgcCrypto_ASN1` reads DER, decodes PEM and Base64, and parses RSA/EC private and public keys out of DER. `sgcCrypto_DER` is the writer, used internally by `sgcCrypto_X509_Gen` and by the RSA key export functions on the Public Key Cryptography page; most applications will not call it directly.

One Time Passwords

HOTP and TOTP are the codes a phone authenticator app shows. Both functions live in `sgcCrypto_OTP` and are built on HMAC (see Hashing & Message Authentication).

HOTP (counter based)

The code depends on a shared secret and a counter that advances by one on every use. The client and the server must stay in step: if the client generates codes the server never sees (a button pressed with no network connection, for example), the counters drift apart and every code the client shows afterward fails until the server resynchronizes. That drift is the main reason TOTP replaced HOTP for most consumer authenticator use.

TOTP (time based)

The counter in HOTP is replaced with the current Unix time divided into fixed steps, 30 seconds by default, so the client and the server only need roughly synchronized clocks rather than a synchronized counter. This is what every authenticator app implements.

Verifying a code

`aWindow` accepts codes from that many steps before and after the current one, which absorbs a small clock skew or a user who was slow to type the code. The comparison is constant time internally, so it does not leak how many digits of a guess were correct.

Provisioning a secret

An authenticator app is set up by scanning a QR code that encodes an `otpauth://` URI. The secret inside it is base32 encoded (see `sgcBase32Encode` on the Cryptography overview page):

Fuzzy Hashing (TLSH)

Every hash on the Hashing & Message Authentication page is built so that a single changed bit produces a completely unrelated digest. TLSH (Trend Micro Locality Sensitive Hash) is built to do the opposite: similar inputs produce similar digests, and `sgcTLSH_Diff` scores how far apart two digests are. It answers "how alike are these two files" rather than "are these two files identical", which is what makes it useful for clustering and near duplicate detection, malware family triage, and finding documents that were copied and lightly edited.

TLSH is not a security hash. Do not use it for integrity or authentication.

It is not collision resistant, and an attacker who controls the input can deliberately steer the digest. Every function is in `sgcCrypto_TLSH`.

Computing a digest

The input must be at least 50 bytes (`cTLSH_MinDataLength`) and varied enough to fill more than half the internal buckets; a digest is 72 characters (`cTLSH_DigestLength`), starting with the version prefix `T1`. An input that is too short or too uniform (a long run of one repeated byte, for example) does not have enough signal to characterize, and the functions below return an empty string rather than a meaningless digest.

A large file does not need to be held in memory at once; feed it through the streaming interface in blocks, or pass a stream directly to `sgcTLSH_Stream`:

Comparing two digests

Zero (`cTLSH_IdenticalDiff`) means the same digest; the score grows with how different the two inputs are, with no fixed upper bound. There is no universal threshold, calibrate it against your own corpus, but as a starting point a distance under roughly 30 usually indicates the same or nearly the same content, while higher values indicate progressively less relation. `aLengthDiff` folds the difference in input length into the score; turn it off when comparing content that is legitimately padded or truncated, such as log files that were rotated at different sizes. The function returns -1 if either digest is malformed.

WinZip AE-2 Archive Encryption

WinZip AE-1 and AE-2 are the AES encryption schemes used inside .zip archives, replacing the original, weak PKZIP 2.0 stream cipher. Every function is in `sgcCrypto_Zip_AE2` and works on one archive entry at a time; building the surrounding .zip container itself is outside the scope of this unit.

Layout

Key material comes from PBKDF2-HMAC-SHA1 over the password, 1000 iterations, producing an encryption key, an authentication key and a 2-byte password verifier in one call. The salt length and the key length both depend on the AES strength chosen for the entry.

Encrypting and decrypting an entry

The 2-byte verifier catches most wrong passwords immediately, and the 10-byte HMAC-SHA1 authentication code that `aAuthCode` carries catches the rest, along with any tampering with the encrypted contents.

What AE-2 does not protect

The authentication code covers only the file contents of one entry. It does not cover the file name, the timestamps, or the archive layout, so an attacker who cannot read the contents can still rename, reorder or drop entries in the archive without detection. It also does not protect against a weak password: the key derivation uses only 1000 PBKDF2 rounds, far fewer than Argon2id or scrypt would use, because the format predates those. An AE-2 archive is only as strong as the password chosen for it.

The extra field

A .zip entry marks itself as AE encrypted by setting its compression method to 99 and adding a small extra field (header ID `0x9901`) that records the real compression method underneath, the AE version (1 or 2) and the AES strength, which is what a reader needs before it can even attempt a password.

Two convenience functions pack and unpack the salt / verifier / ciphertext / authentication code layout that follows the local file header, for callers assembling or reading a full entry rather than only its cryptography:

FIPS Mode

sgcCrypto has an optional compile time switch, `SGC_CRYPTO_FIPS`, that leaves only FIPS-approved algorithms in the library. It is meant for projects that must not use algorithms outside the FIPS and NIST SP publications. This page explains what the switch does, how to turn it on, what it keeps, what it removes, and what it does not cover.

What FIPS means here

FIPS 140-3 is a certification of a cryptographic *module*, not of an algorithm. An accredited laboratory tests one specific build of a product under NIST's Cryptographic Module Validation Program (CMVP), and NIST publishes a certificate for that build. Using an approved algorithm such as AES does not make a program validated.

sgcCrypto is not a FIPS 140-3 validated module, and turning on the switch is not a validation claim. What the switch does is narrower:

- It removes every algorithm that is not FIPS-approved, so none of them can be compiled into your application.
- It enforces a few parameter rules at run time, for example a minimum RSA key size for signing.

The algorithms that stay follow the FIPS and NIST SP publications listed in the table below.

If your project has a formal FIPS 140-3 requirement, its cryptography must go through a validated module, for example Windows CNG with the system running in FIPS mode. The switch helps keep algorithms that are not approved out of your code. It does not replace a validated module.

Turning it on

The switch is off by default. Without it nothing changes, and every algorithm listed on the Cryptography page is available. To turn it on, do one of the following:

- Open `sgcVer.inc` and change `{.$DEFINE SGC_CRYPTO_FIPS}` to `{$DEFINE SGC_CRYPTO_FIPS}`.
- Add `SGC_CRYPTO_FIPS` to the conditional defines of the project when you compile the sgcWebSockets sources.

Then rebuild. The switch only affects code compiled with the define, so precompiled packages that were built without it are not affected.

With the switch on, the removed functions no longer exist. Code that still calls one of them fails to compile with an "Undeclared identifier" error. This is intentional: nothing that is not approved can end up in the binary by accident, and the compiler points at every place that needs a different algorithm.

What stays

Certificates, one time passwords and the encoding helpers stay as well, because they are built only from the approved algorithms or are not cryptographic algorithms at all.

Family	Kept	Standard
AES	AES-128/192/256 in ECB, CBC, CFB, OFB, CTR, CTS, GCM and CCM mode, AES Key Wrap (KW and KWP), AES-CMAC and AES-GMAC	FIPS 197, SP 800-38A/B/C/D/F
Hashing	SHA-1 (only where still allowed, never for new signatures), SHA-224, SHA-256, SHA-384, SHA-512, SHA3-224/256/384/512, SHAKE128, SHAKE256, cSHAKE and KMAC	FIPS 180-4, FIPS 202, SP 800-185
Message authentication	HMAC-SHA1, HMAC-SHA256, HMAC-SHA384 and HMAC-SHA512	FIPS 198-1
Key derivation	PBKDF2, HKDF and the ANSI X9.63 KDF	SP 800-132 (PBKDF2), SP 800-56C (HKDF), SP 800-135 (ANSI X9.63 KDF)
RSA	RSA key generation, PKCS#1 v1.5 and PSS signatures, RSA-OAEP encryption	FIPS 186-5, SP 800-56B
ECDSA and ECDH	Signatures and key agreement on NIST P-256, P-384 and P-521 (P-224 is not implemented by sgcCrypto)	FIPS 186-5, SP 800-56A
EdDSA	Ed25519 and Ed448 signatures	FIPS 186-5
Post-quantum cryptography	ML-KEM, ML-DSA and SLH-DSA	FIPS 203, FIPS 204, FIPS 205

Certificates & PKI	X.509 certificate parsing, chain verification and CRL checks, and certificate or CSR generation signed with the algorithms above	RFC 5280, RFC 2986
One time passwords	HOTP and TOTP, built on HMAC	RFC 4226, RFC 6238
Encoding & utilities	Hex, Base32 and Base64 encoding, constant time compare, secure zero	RFC 4648

What is removed

Removed	Why
ChaCha20, XChaCha20, Salsa20, XSalsa20, Poly1305, ChaCha20-Poly1305 and XChaCha20-Poly1305	Not NIST algorithms.
X25519 and X448 key exchange	Not approved for key agreement in SP 800-56A.
The X25519 + ML-KEM-768 hybrid, and ECIES	Both are built on X25519.
secp256k1 and the Brainpool curves, and BIP-340 Schnorr signatures	Not NIST recommended curves, and BIP-340 is not a FIPS 186-5 signature scheme. FIPS mode keeps only the NIST P curves.
Argon2 and scrypt	Not NIST password key derivation functions.
BLAKE2b, BLAKE2s, SipHash and TLSH	Not NIST algorithms.
MD4, MD5, HMAC-MD5, RIPEMD-160 and DES (the `sgcCrypto_Legacy` unit)	Obsolete, or never approved by NIST.
KDF1 and KDF2	Defined in ISO 18033-2, not in a NIST publication. The ANSI X9.63 KDF stays.
RSA PKCS#1 v1.5 encryption	Disallowed by SP 800-131A. PKCS#1 v1.5 signatures stay.
WinZip AES (AE-1 and AE-2)	Not a NIST format.

For most removed algorithms an approved replacement is in the first table: AES-GCM instead of ChaCha20-Poly1305, ECDH on P-256 or P-384 instead of X25519, PBKDF2 instead of Argon2 or scrypt, SHA-256 or SHA-3 instead of BLAKE2, and RSA-OAEP instead of PKCS#1 v1.5 encryption.

Rules checked at run time

Some rules depend on the values passed to a function, so the compiler cannot enforce them. With the switch on, sgcCrypto checks them when the function runs, and each one raises an exception whose message starts with `sgcCrypto FIPS:`

- Creating a signature with SHA-1, with RSA or ECDSA, including signing a certificate or a CSR. Verifying an existing SHA-1 signature is still allowed.
- Signing with an RSA key smaller than 2048 bits.
- RSA key generation runs the extra key checks from FIPS 186-5 and a pairwise consistency test on the new key pair.
- Generating an EC key from a seed needs a seed at least 64 bits longer than the curve order, for example at least 320 bits for P-256.
- PBKDF2 with a salt shorter than 16 bytes, or with a derived key shorter than 14 bytes.
- AES-GCM with an empty IV.

Where the random bytes come from

Keys, nonces and salts need random bytes. `sgcRandomBytes` always reads them from the operating system, in both modes, and never runs a generator of its own on top. The switch only changes which system source is used:

Platform	With SGC_CRYPTO_FIPS	Without SGC_CRYPTO_FIPS
Windows	BCryptGenRandom (CNG, system preferred RNG), with no fallback	BCryptGenRandom, falling back to RtlGenRandom and then CryptGenRandom when BCryptGenRandom is not available
Linux and Android	The getrandom system call	/dev/urandom

macOS and iOS	SecRandomCopyBytes from the Security framework	/dev/urandom

In both modes, if no source can be read, an exception is raised instead of returning weak bytes. Whether the operating system generator counts as FIPS depends on the operating system and its configuration, not on sgcCrypto.

What the switch does not cover

The switch applies to the sgcCrypto units only. Other parts of sgcWebSockets can still use algorithms that are not approved:

- MD5 in TURN long-term credentials, in the Pusher API signature and in the old hixie-76 WebSocket handshake.
- MD4 and DES in NTLM authentication.
- The cipher suites and key exchange that OpenSSL negotiates for TLS and QUIC connections, such as ChaCha20-Poly1305 or X25519. These are controlled by your OpenSSL configuration, not by this switch.
- Amazon CloudFront signed URLs, which require RSA with SHA-1 by design. They raise an exception when the switch is on.

If the whole application must stay within approved algorithms, review these areas as well.

See also

- Cryptography
- Post-Quantum Cryptography

IoT

The Internet of things (IoT) refers to the concept of extending Internet connectivity beyond conventional computing platforms such as personal computers and mobile devices, and into any range of traditionally "dumb" or non-internet-enabled physical devices and everyday objects. Embedded with electronics, Internet connectivity, and other forms of hardware (such as sensors), these devices can communicate and interact with others over the Internet, and they can be remotely monitored and controlled.

sgcWebSockets package implements the following IoT clients:

1. Amazon AWS IoT: AWS IoT provides secure, bi-directional communication between Internet-connected devices such as sensors, actuators, embedded micro-controllers, or smart appliances and the AWS Cloud. This enables you to collect telemetry data from multiple devices, and store and analyze the data. You can also create applications that enable your users to control these devices from their phones or tablets.

2. Azure IoT Hub: IoT Hub is a managed service, hosted in the cloud, that acts as a central message hub for bi-directional communication between your IoT application and the devices it manages. You can use Azure IoT Hub to build IoT solutions with reliable and secure communications between millions of IoT devices and a cloud-hosted solution backend.

IoT Amazon MQTT Client

What Is AWS IoT?

AWS IoT provides secure, bi-directional communication between Internet-connected devices such as sensors, actuators, embedded micro-controllers, or smart appliances and the AWS Cloud. This enables you to collect telemetry data from multiple devices, and store and analyze the data. You can also create applications that enable your users to control these devices from their phones or tablets.

Message broker

Provides a secure mechanism for devices and AWS IoT applications to publish and receive messages from each other. You can use either the MQTT protocol directly or MQTT over WebSocket to publish and subscribe.

The AWS IoT message broker is a publish/subscribe broker service that enables the sending and receiving of messages to and from AWS IoT. When communicating with AWS IoT, a client sends a message addressed to a topic like Sensor/temp/room1.

The message broker, in turn, sends the message to all clients that have registered to receive messages for that topic. The act of sending the message is referred to as publishing. The act of registering to receive messages for a topic filter is referred to as subscribing.

The topic namespace is isolated for each AWS account and region pair. For example, the Sensor/temp/room1 topic for an AWS account is independent from the Sensor/temp/room1 topic for another AWS account. This is true of regions, too. The Sensor/temp/room1 topic in the same AWS account in us-east-1 is independent from the same topic in us-east-2. AWS IoT does not support sending and receiving messages across AWS accounts and regions.

The message broker maintains a list of all client sessions and the subscriptions for each session. When a message is published on a topic, the broker checks for sessions with subscriptions that map to the topic. The broker then forwards the publish message to all sessions that have a currently connected client.

MQTT Client

TsgcIoTAmazon_MQTT_Client is the component used to connect to AWS IoT. One client can connect to only one device. The client connects using the plain MQTT protocol and authenticates using an X.509 Client Certificate.

In order to connect to AWS IoT, the client needs the following properties:

Amazon.ClientId: identification of client, optional.
Amazon.Endpoint: server name where MQTT client will connect.
Amazon.Port: by default uses port 8883. If port is 443, uses ALPN automatically to connect (Requires custom Indy version).

AWS IoT Core supports devices and clients that use the MQTT and the MQTT over WebSocket Secure (WSS) protocols to publish and subscribe to messages. The following table lists the protocols that the AWS IoT device endpoints support and the authentication methods and ports they use.

Protocol	Authentication	Port	ALPN Protocol Name
MQTT over WebSocket	Signature Version 4	443	
MQTT over WebSocket	Custom Authentication	443	

MQTT	X.509 client certificate	443	x-amzn-mqtt-ca
MQTT	X.509 client certificate	8883	
MQTT	Custom Authentication	443	mqtt

Certificates Authentication

You need to create certificates in your Amazon AWS console and set the path where they are stored.

Using **OpenSSL** as IOHandler you must set the certificate in the following paths

Certificate.Enabled: set to True if you want to use certificates.
Certificate.CertFile: path to X.509 client certificate.
Certificate.KeyFile: path to X.509 client key file.

Using **SChannel** as IOHandler, first convert the PEM Certificate + Key to a PFX certificate. This requires OpenSSL binaries:

```
openssl pkcs12 -inkey 884ccf73ff-private.pem.key -in 884ccf73ff-certificate.pem.crt -export -out 884ccf73ff-certi
```

Then set the following paths (there is no need to set the key file because it is already included in the certificate).

Certificate.Enabled: set to True if you want to use certificates.
Certificate.CertFile: path to PFX certificate

Using **NativeTLS** as IOHandler (**Certificate.IOHandler** = iohNativeTLS), no OpenSSL library is needed for the TLS connection. See Native TLS.

Certificate.Enabled: set to True if you want to use certificates.
Certificate.CertFile: path to the PEM certificate, or to a .pfx / .p12 file that holds the certificate and the key (opened with Certificate.Password).
Certificate.KeyFile: path to the PEM key file, not needed with a .pfx / .p12 file.
Certificate.NativeTLS_Options: CipherSuites, Groups, UseSystemRoots, SessionTickets and TicketLifetime of the native engine. They are copied to the inner WebSocket client before it connects.

SignatureV4 Authentication

You need to create a user in your Amazon AWS console and save the Access and Secret keys, which will be used to sign the WebSocket request.

SignatureV4.Enabled: set to True if you want to use this type of authentication.
SignatureV4.Region: the region where your device is located (example: us-east-1).
SignatureV4.AccessKey: the access key created in your Amazon console or obtained as a temporary credential.
SignatureV4.SecretKey: the secret key created in your amazon console or get as temporary credential
SignatureV4.SessionToken: (conditional) if you are using Temporary Security Credentials, set here the security token.
OpenSSL_Options: configuration of the openSSL libraries. When **sgcCryptoBackend** selects the native backend (or the library is built with SGC_NATIVE_CRYPTO), the Signature V4 hashes run on the native sgcCrypto units, OpenSSL is not loaded for signing and these options are ignored. See Running without OpenSSL.
APIVersion: allows defining which OpenSSL API will be used.
oslAPI_1_0: uses API 1.0 OpenSSL, it's latest supported by Indy

oslAPI_1_1: uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).
oslAPI_3_0: uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).

LibPath: here you can configure where are located the openSSL libraries
oslpNone: this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.
oslpDefaultFolder: sets automatically the openSSL path where the libraries should be located for all IDE personalities.
oslpCustomFolder: if this is the option selected, define the full path in the property LibPath-Custom.

LibPathCustom: when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.

UnixSymLinks: enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
oslsSymLinksDefault: by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).
oslsSymLinksLoadFirst: Load SymLinks and do before trying to load the version libraries.
oslsSymLinksLoad: Load SymLinks after trying to load the version libraries.
oslsSymLinksDontLoad: don't load the SymLinks.

**SignatureV4 requires Indy 10.5.7+*

Custom Authentication

Custom authentication enables you to define how to authenticate and authorize clients by using authorizer resources. The device passes credentials in either the request's header fields or query parameters (for MQTT over WebSockets protocols) or in the user name and password field of the MQTT CONNECT message (for the MQTT and MQTT over WebSockets protocols).

CustomAuthentication.Enabled: set to True if you want to use this type of Authentication.
CustomAuthentication.Parameters: set here the query parameters which will be passed to the server (by default is /mqtt)
CustomAuthentication.Headers: here you can put the custom header fields.
CustomAuthentication.WebSockets: if set to true, the connection will work over WebSocket protocol, otherwise will work over plain TCP.

MQTTAuthentication.Enabled: if you need to pass the username/password in the mqtt connection, enable this property
MQTTAuthentication.Username: username of the mqtt connection
M**QTTAuthentication.Password:** secret of the mqtt connection.

Client can send optionally a ClientId to identifiy client connection, then other clients can subscribe to receive a notification every time this client has connected, subscribed, disconnected...

Authorization

If you can't connect using port 8883 and use TCP as transport (which is the default), amazon takes "AWS IoT Core policy" to provide or not authorization to clients and subscriptions. Most probably you must authorize your client id. Enter in your Amazon AWS console, go to IoT Core and access the menu "Secure/Policies", there select the policy attached to your IoT Thing and check at the end how connection is configured. Example:

```
{
  "Effect": "Allow",
  "Action": [
    "iot:Connect"
  ],
  "Resource": [
    "arn:aws:iot:us-east-1:222178873557:client/sdk-java",
    "arn:aws:iot:us-east-1:222178873557:client/basicPubSub",
    "arn:aws:iot:us-east-1:222178873557:client/sdk-nodejs-*"
  ]
```

}

This configuration means that only clients with ID: sdk-java, basicPubSub and sdk-nodejs-* will be allowed to connect. Change accordingly and try again.
If it still doesn't work, enable log and check in cloudwatch the reason why you can't connect.

Other properties

MQTTHeartBeat: if enabled attempts to keep alive MQTT connection sending a ping every x seconds.

Interval: number of seconds between each ping.

MQTTAuthentication: if enabled includes in MQTT connection the username and password

UserName: name of the user
Password: secret string

WatchDog: if enabled, when an unexpected disconnection is detected, tries to reconnect to the server automatically.

Interval: seconds before reconnection attempts.

Attempts: maximum number of reconnection attempts; zero means unlimited.

LogFile: if enabled, saves socket messages to a log file (useful for debugging). The access to log file is not thread safe if it's accessed from several threads.

Enabled: if enabled every time a message is received and sent by socket it will be saved on a file.

FileName: full path to the filename.

Implementation

Amazon MQTT implementation is based on MQTT version 3.1.1 but it deviates from the specification as follows:

- In AWS IoT, subscribing to a topic with Quality of Service (QoS) 0 means a message is delivered zero or more times. A message might be delivered more than once. Messages delivered more than once might be sent with a different packet ID. In these cases, the DUP flag is not set.
- AWS IoT does not support publishing and subscribing with QoS 2. The AWS IoT message broker does not send a PUBACK or SUBACK when QoS 2 is requested.
- When responding to a connection request, the message broker sends a CONNACK message. This message contains a flag to indicate if the connection is resuming a previous session. The value of this flag might be incorrect if two MQTT clients connect with the same client ID simultaneously.
- When a client subscribes to a topic, there might be a delay between the time the message broker sends a SUBACK and the time the client starts receiving new matching messages.
- The MQTT specification provides a provision for the publisher to request that the broker retain the last message sent to a topic and send it to all future topic subscribers. AWS IoT does not support retained messages. If a request is made to retain messages, the connection is disconnected.
- The message broker uses the client ID to identify each client. The client ID is passed in from the client to the message broker as part of the MQTT payload. Two clients with the same client ID are not allowed to be connected concurrently to the message broker. When a client connects to the message broker using a client ID that another client is using, a CONNACK message is sent to both clients and the currently connected client is disconnected.
- On rare occasions, the message broker might resend the same logical PUBLISH message with a different packet ID.
- The message broker does not guarantee the order in which messages and ACK are received.

Connect to AWS IoT

First, you must sign in your AWS console, register a new device and create a X.509 certificate for this device. Once is done, you can create a new TsgcIoTAmazon_MQTT_Client and connect to AWS IoT Server. For example:

```
oClient = new TsgcIoTAmazon_MQTT_Client();
oClient->Amazon->Endpoint = "a2ohgdjqitsmij-ats.iot.us-west-2.amazonaws.com";
oClient->Amazon->ClientId = "sgcWebSockets";
oClient->Certificate->CertFile = "amazon-certificate.pem.crt";
oClient->Certificate->KeyFile = "amazon-private.pem.key";
oClient->OnMQTTConnect = OnMQTTConnectEvent;
oClient->Active = true;

void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const TmqttConnReturnCode
ReturnCode)
{
  ShowMessage("Connected to AWS");
}
```

Topics

The message broker uses topics to route messages from publishing clients to subscribing clients. The forward slash (/) is used to separate topic hierarchy. The following table lists the wildcards that can be used in the topic filter when you subscribe. # Must be the last character in the topic to which you are subscribing. Works as a wildcard by matching the current tree and all subtrees.
For example, a subscription to Sensor/# receives messages published to Sensor/, Sensor/temp, Sensor/temp/room1, but not the messages published to Sensor.
+ Matches exactly one item in the topic hierarchy. For example, a subscription to Sensor/+/room1 receives messages published to Sensor/temp/room1, Sensor/moisture/room1, and so on.

```
oClient = new TsgcIoTAmazon_MQTT_Client();
...
oClient->OnSubscribe = OnSubscribeEvent;

vPacketIdentifier = oClient->Subscribe("Sensor/moisture/room1");

void OnMQTTSubscribe(TsgcWSConnection *Connection, Word aPacketIdentifier, TsgcWSSUBACKS *aCodes)
{
  if (vPacketIdentifier == aPacketIdentifier)
  {
    ShowMessage("Subscribed to topic Sensor/moisture/room1");
  }
}

// Client, can send a message using Publish method.
oClient->Publish("Sensor/moisture/room1", "{"\"temp"\"=10}");

// Messages received from server, are dispatched OnMQTTPublishEvent.
// For extended payload access (string, bytes or stream), use OnMQTTPublishEx.
void OnMQTTPublish(TsgcWSConnection *Connection, string aTopic, string aText)
{
  DoLog("Received Message: " + aTopic + " " + aText);
}
```

Reserved Topics

Following methods are used to subscribe / publish to reserved topics.

Subscribe_ClientConnected(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID connects to AWS IoT
Subscribe_ClientDisconnected(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID disconnects to AWS IoT
Subscribe_ClientSubscribed(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID subscribes to an MQTT topic
Subscribe_ClientUnSubscribed(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID unsubscribes to an MQTT topic

Publish_Rule(const aRuleName, aText: String): A device or an application publishes to this topic to trigger rules directly

Publish_DeleteShadow(const aThingName, aText: String): A device or an application publishes to this topic to delete a shadow
Subscribe_DeleteShadow(const aThingName: String): A device or an application subscribe to this topic to delete a shadow
Subscribe_ShadowDeleted(const aThingName: String): The Device Shadow service sends messages to this topic when a shadow is deleted
Subscribe_ShadowRejected(const aThingName: String): The Device Shadow service sends messages to this topic when a request to delete a shadow is rejected
Publish_ShadowGet(const aThingName, aText: String): An application or a thing publishes an empty message to this topic to get a shadow
Subscribe_ShadowGet(const aThingName: String): An application or a thing subscribe to this topic to get a shadow
Subscribe_ShadowGetAccepted(const aThingName: String): The Device Shadow service sends messages to this topic when a request for a shadow is made successfully
Subscribe_ShadowGetRejected(const aThingName: String): The Device Shadow service sends messages to this topic when a request for a shadow is rejected
Publish_ShadowUpdate(const aThingName, aText: String): A thing or application publishes to this topic to update a shadow
Subscribe_ShadowUpdateAccepted(const aThingName: String): The Device Shadow service sends messages to this topic when an update is successfully made to a shadow
Subscribe_ShadowUpdateRejected(const aThingName: String): The Device Shadow service sends messages to this topic when an update to a shadow is rejected
Subscribe_ShadowUpdateDelta(const aThingName: String): The Device Shadow service sends messages to this topic when a difference is detected between the reported and desired sections of a shadow
Subscribe_ShadowUpdateDocuments(const aThingName: String): AWS IoT publishes a state document to this topic whenever an update to the shadow is successfully performed

Persistent Sessions

A persistent session represents an ongoing connection to an MQTT message broker. When a client connects to the AWS IoT message broker using a persistent session, the message broker saves all subscriptions the client makes during the connection. When the client disconnects, the message broker stores unacknowledged QoS 1 messages and new QoS 1 messages published to topics to which the client is subscribed. When the client reconnects to the persistent session, all subscriptions are reinstated and all stored messages are sent to the client at a maximum rate of 10 messages per second.

You create an MQTT persistent session setting the **cleanSession** parameter to False **OnMQTTBeforeConnect** event. If no session exists for the client, a new persistent session is created. If a session already exists for the client, it is resumed.

Devices need to look at the **Session** attribute in the **OnMQTTConnect** event to determine if a persistent session is present. If **Session is True**, a persistent session is present and stored messages are delivered to the client. If **Session is False**, no persistent session is present and the client must re-subscribe to its topic filters.

Persistent sessions have a default expiry period of 1 hour. The expiry period begins when the message broker detects that a client disconnects (MQTT disconnect or timeout). The persistent session expiry period can be increased through the standard limit increase process. If a client has not resumed its session within the expiry period, the session is terminated and any associated stored messages are discarded. The expiry period is approximate, sessions might be persisted for up to 30 minutes longer (but not less) than the configured duration.

Temporary Credentials

AWS IoT Core can work with Temporary Credentials obtained through Identity Pools, there are 2 types of Identities:

- **UnAuthenticated:** only requires to set the policy type in the IAM
- **Authenticated:** requires to set the policy type in IAM and AWS IoT Core policies

Unauthenticated

If you are using Unauthenticated credentials, just attach the policy in the UnAuthenticated Role automatically created in the IAM menu. Then configure the client setting the Access, Secret Key and Token returned by Cognito service.
Find below a code in .NET to get unauthenticated credentials

```
CognitoAWSCredentials credentials = new CognitoAWSCredentials(
    "us-east-1:cc3c9c48-646d-44ef-bfd5-0c5fb2f0882f", // Identity pool ID
    Amazon.RegionEndpoint.USEast1 // Region
);

var identityPoolId = credentials.GetCredentialsAsync();

AmazonCognitoIdentityClient cognitoClient = new AmazonCognitoIdentityClient(
    credentials, // the anonymous credentials
    Amazon.RegionEndpoint.USEast1 // the Amazon Cognito region
);

GetIdRequest idRequest = new GetIdRequest();
idRequest.AccountId = "222178873557";
idRequest.IdentityPoolId = "us-east-1:cc3c9c48-646d-44ef-bfd5-0c5fb2f0882f";

GetIdResponse idResp = cognitoClient.GetId(idRequest);

string AccessKey = identityPoolId.Result.AccessKey;
string SecretKey = identityPoolId.Result.SecretKey;
string SessionToken = identityPoolId.Result.Token;

string IdentityId = idResp.IdentityId;
```

Authenticated

Authenticated credentials, requires attaching the policy in the Authenticated Role automatically created in the IAM menu and attach the policy of the user in AWS IoT Core policies.
So create a new policy in the IoT Core policies menu and every time a new user authenticates, attach this policy to this user.
You can use the following command of AWS to attach a policy or create a lambda function.

```
aws iot attach-policy --policy-name PolicyName --target us-east-1:XXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
```

Device Provisioning

The Fleet Provisioning service supports the following MQTT API operations:

- **CreateCertificateFromCsr:** Creates a certificate from a certificate signing request (CSR). AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA). The new certificate has a PENDING_ACTIVATION status. When you call RegisterThing to provision a thing with this certificate, the certificate status changes to ACTIVE or INACTIVE as described in the template.
- **CreateKeysAndCertificate:** Creates new keys and a certificate. AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA). The new certificate has a PENDING_ACTIVATION status. When you call RegisterThing to provision a thing with this certificate, the certificate status changes to ACTIVE or INACTIVE as described in the template.
- **RegisterThing:** Provisions a thing using a pre-defined template.

CreateCertificateFromCsr

Use the method CreateCertificateFromCsr passing the CertificateSigningRequest as a parameter to create the certificate. In order to receive the response to this request, first subscribe to the following methods: SubscribeCreateCertificateFromCsrResponse and SubscribeCreateCertificateFromCsrError

CreateKeysAndCertificate

Use the method CreateKeysAndCertificate to create a new certificate and keys. In order to receive the response to this request, first subscribe to the following methods SubscribeCreateKeysAndCertificateResponse and SubscribeCreateKeysAndCertificateError

RegisterThing

Use the method RegisterThing to register a new thing passing as a parameter the Template Name and the Payload in JSON format. In order to receive the response to this request, first subscribe to the following methods SubscribeRegisterThingResponse and SubscribeRegisterThingError.

IoT Azure MQTT Client

What is Azure IoT Hub?

IoT Hub is a managed service, hosted in the cloud, that acts as a central message hub for bi-directional communication between your IoT application and the devices it manages. You can use Azure IoT Hub to build IoT solutions with reliable and secure communications between millions of IoT devices and a cloud-hosted solution backend. You can connect virtually any device to IoT Hub.

IoT Hub supports communications both from the device to the cloud and from the cloud to the device. IoT Hub supports multiple messaging patterns such as device-to-cloud telemetry, file upload from devices, and request-reply methods to control your devices from the cloud. IoT Hub monitoring helps you maintain the health of your solution by tracking events such as device creation, device failures, and device connections.

IoT Hub's capabilities help you build scalable, full-featured IoT solutions such as managing industrial equipment used in manufacturing, tracking valuable assets in healthcare, and monitoring office building usage.

Message broker

IoT Hub gives you a secure communication channel for your devices to send data. IoT Hub and the device SDKs support the following protocols for connecting devices:

- **MQTT**
- **MQTT** over **WebSockets**

Multiple authentication types support a variety of device capabilities:

- **SAS** token-based authentication to quickly get started with your IoT solution.
- Individual **X.509 certificate** authentication for secure, standards-based authentication.

MQTT Client

TsgcIoTAzure_MQTT_Client is the component used to connect to Azure IoT. One client can connect to only one device. The client connects using the plain MQTT protocol and authenticates using SAS / X.509 Client Certificate.

In order to connect to Azure IoT Hub, client needs the following properties:

Azure.IoTHub: server name where MQTT client will connect.
Azure.DeviceId: name of device in azure IoT Hub.

Azure allows multiple authentication types, by default uses SAS tokens.

SAS Authentication

SAS.Enabled: enable if authentication uses SAS.
SAS.SecretKey: the SAS Token from your Azure IoT Account.
SAS.KeyName: the Shared Access Key Name.
SAS.Expiry: set the number of minutes before SAS Token expires. Default value is 1440 (24 hours).

If you have a connection string, you can read the connection string values automatically using the method **ReadConnectionString**. Example:

```
ReadConnectionString('HostName=yourhub.azure-
devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=Yj7RRPnkSDTv+UCFLgwIP/
FrbDymZv4qVAIoTLHUFR8=');
```

X509 Certificates

Using **OpenSSL** as IOHandler

Certificate.Enabled: enable if authentication uses certificates.
Certificate.CertFile: path to X.509 client certificate.
Certificate.KeyFile: path to X.509 client key file.
Certificate.Password: if certificate has a password set here.
Version: TLS version, by default uses TLS 1.0

Using **SChannel** as IOHandler

Certificate.Enabled: enable if authentication uses certificates.
Certificate.CertFile: path to PFX certificate (first the certificate must be converted to PFX). Read More.
Certificate.Password: if certificate has a password set here.
Version: TLS version, by default uses TLS 1.0

Using **NativeTLS** as IOHandler

Certificate.Enabled: enable if authentication uses certificates.
Certificate.CertFile: path to the PEM certificate, or to a .pfx / .p12 file that holds the certificate and the key.
Certificate.KeyFile: path to the PEM key file, not needed with a .pfx / .p12 file.
Certificate.Password: if certificate has a password set here.
Certificate.NativeTLS_Options: CipherSuites, Groups, UseSystemRoots, SessionTickets and TicketLifetime of the native engine. They are copied to the inner WebSocket client and to the internal HTTP client. See Native TLS.

Other properties:

MQTTHeartBeat: if enabled attempts to keep alive MQTT connection sending a ping every x seconds.

Interval: number of seconds between each ping.

WatchDog: if enabled, when an unexpected disconnection is detected, tries to reconnect to the server automatically.

Interval: seconds before reconnection attempts.

Attempts: maximum number of reconnection attempts; zero means unlimited.

LogFile: if enabled, saves socket messages to a log file (useful for debugging). The access to log file is not thread safe if it's accessed from several threads.

Enabled: if enabled every time a message is received and sent by socket it will be saved on a file.

FileName: full path to the filename.

Azure MQTT implementation is based on MQTT version 3.1.1 but it deviates from the specification as follows:

- IoT Hub does not support QoS 2 messages. If a device app publishes a message with QoS 2, IoT Hub closes the network connection.
- IoT Hub does not persist Retain messages. If a device sends a message with the RETAIN flag set to 1, IoT Hub adds the x-opt-retain application property to the message. In this case, instead of persisting the retain message, IoT Hub passes it to the backend app.
- IoT Hub only supports one active MQTT connection per device. Any new MQTT connection on behalf of the same device ID causes IoT Hub to drop the existing connection.

Connect to Azure IoT Hub

First, you must sign in to your Azure account, register a new device, and create an authentication method for this device. Once that is done, you can create a new TsgcIoTAzure_MQTT_Client and connect to Azure IoT Hub.

For example:

```
TsgcIoTAzure_MQTT_Client *oClient = new TsgcIoTAzure_MQTT_Client();
oClient->Azure->IoTHub = "youriothub.azure-devices.net";
oClient->Azure->DeviceId = "YourDeviceId";
oClient->SAS->Enabled = true;
oClient->SAS->SecretKey = "YourSecretKey";
oClient->OnMQTTConnect = OnMQTTConnectEvent;
oClient->Active = true;

private void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const TmqttConnReturnCode *ReturnCoc
{
  ShowMessage("Connected to Azure IoT Hub");
}
```

Device To Cloud

When sending information from the device app to the solution back end, IoT Hub exposes the following options:

1. **Device-to-cloud messages** for time series telemetry and alerts.

```
oClient->Send_DeviceToCloud("{\"temp\": 10}", azuIoTQoS1);
```

You can send key-value properties using a TStringList, just fill the TStringList with the desired message properties and pass these as argument.

```
TStringList *oProperties = new TStringList;
try
{
  oProperties->AddPair("prop_name1", "prop_value1");
  oProperties->AddPair("prop_name2", "prop_value2");
  oClient->Send_DeviceToCloud("{\"temp\": 10}", oProperties, azuIoTQoS1);
}
__finally
{
  oProperties->Free();
}
```

If you need to set the **ContentType** and **ContentEncoding** of the message, you must add these values to the Properties List. The names of these properties are defined by Azure.

Name	Value
$.ct	application/json
$.ce	utf-8

```
TStringList *oProperties = new TStringList();
try
{
  oProperties->AddPair("$.ct", "application/json");
  oProperties->AddPair("$.ce", "utf-8");
  oClient->Send_DeviceToCloud("{\"temp\": 10}", oProperties, azuIoTQoS1);
}
__finally
{
  oProperties->Free();
}
```

2. **Device twin's reported properties** for reporting device state information such as available capabilities, conditions, or the state of long-running workflows. For example, configuration and software updates.

```
oClient->Set_DeviceTwinsProperties("1", "{\"sgc\":1}");
```

Cloud To Device

IoT Hub provides three options for device apps to expose functionality to a back-end app:

1. **Direct methods** for communications that require immediate confirmation of the result. Direct methods are often used for interactive control of devices such as turning on a fan.

```
oClient->Subscribe_DirectMethod();
```

You can respond to public methods using the following method.

```
oClient->RespondPublicMethod(RequestId, Status, "Your Response", azuIoTQoS1);
```

2. **Twin's desired properties** for long-running commands intended to put the device into a certain desired state. For example, set the telemetry send interval to 30 minutes. You can get properties using the following method.

```
oClient->Get_DeviceTwinsProperties("1");
```

3. **Cloud-to-device messages** for one-way notifications to the device app. To get messages, first you must subscribe.

```
oClient->Subscribe_CloudToDevice;
```

Messages are received in the **OnMQTTPublish** event (text payload) or **OnMQTTPublishEx** event (payload as TsgcWSMQTTPublishData with Value, Bytes and Stream properties).

```
void AzureIoTMQTTPublish(TsgcWSConnection *Connection, string aTopic, string aText)
{
  DoLog("Received Message: " + aTopic + " " + aText);
}
```

Upload Files

IoT hub facilitates file uploads from connected devices by providing them with shared access signature (SAS) URIs or X509 certificates.

If you select SAS, you must set the following properties:

- **Azure.IoTHub:** example youriothub.azure-devices.net
- **Azure.DeviceId:** example: myDevice
- **SAS.SecretKey:** example: Yj7RRPnkSDTv+UCFLgwIP/FrbDymZv4qVAIoTLHUFR8=
- **SAS.KeyName:** example: iothubowner
- **SAS.Enabled:** the value must be set to true.

If you select X509 certificates, you must set the following properties:

- **Certificate.CertFile:** the path to your PEM certificate.
- **Certificate.KeyFile:** the path to your PEM key file.
- **Certificate.Enabled:** the value must be set to true.

Use the method **UploadFile** to upload a file to the Azure Servers. If the Overwrite parameter is set to true, it will replace the existing file. If the Overwrite parameter is set to false, it will only upload if the file doesn't exist and will raise an error if it exists (this is the option by default).

```
void UploadFileToAzure()
{
    TOpenDialog* oDialog = new TOpenDialog(NULL);
    try
    {
        if (oDialog->Execute())
```

```
        {
            AnsiString fileName = oDialog->FileName;
            AzureIoT::UploadFile(fileName.c_str());
        }
    }
    __finally
    {
        oDialog->Free();
    }
}
```

Device Provisioning Service

Azure IoT allows you to register devices from code using DPS. Currently, the library supports registering a device passing the Scope Id and Registration Id as parameters.

```
TsgcIoTAzure_MQTT_Client* oClient = new TsgcIoTAzure_MQTT_Client(NULL);
try
{
    oClient->Certificate->CertFile = L"cert.pem";
    oClient->Certificate->KeyFile = L"key.pem";
    oClient->Certificate->Enabled = true;
    TsgcIoT_Azure_OperationRegistrationState* oResponse = new TsgcIoT_Azure_OperationRegistrationState();
    try
    {
        if (oClient->ProvisioningDeviceClient_Register(L"scope_id", L"registration_id", oResponse))
            ShowMessage(L"#Provisioning Register OK: " + oResponse->Status);
        else
            ShowMessage(L"#Provisioning Register Error: " + oResponse->Status);
    }
    __finally
    {
        delete oResponse;
    }
}
__finally
{
    delete oClient;
}
```

Azure IoT Explorer

You can use the **Azure IoT Explorer** application to interact with devices connected to your IoT Hub. You can see the telemetry messages received, the devices registered, and more. The application is free and can be downloaded from:

https://github.com/Azure/azure-iot-explorer/releases

HTTP

The HTTP protocol allows you to fetch resources from servers, such as images and HTML documents. It is a client-server protocol, which means that the client requests from the server the resources it needs.

When a client wants to connect to a server, it follows the next steps:

1. Open a new TCP connection
2. Send a message to the server with the requested data

```
GET / HTTP/1.1
Host: server.com
Accept-Language: en-us
```

3. Read the response sent by the server

```
HTTP/1.1 200 OK
Server: Apache
Content-Length: 120
Content-Type: text/html

...
```

Components

- **HTTP/2:** HTTP/2 (or h2) is a binary protocol that brings push, multiplexing streams and frame control to the web.
- **gRPC:** a contract-based RPC framework that runs over HTTP/2 and uses Protocol Buffers, with a gRPC client component and typed interfaces for Google Cloud services.
- **HTTP/1 Client:** a non-visual component that inherits from the TIdHTTP client component.
- **Server-Sent Events Client:** TsgcSSEClient receives text/event-stream events from a server, reconnects automatically and resumes the stream with Last-Event-ID.
- **OAuth2:** OAuth2 allows third-party applications to receive limited access to an HTTP service.
- **JWT:** JWT allows creating data with optional signature and/or encryption whose payload holds JSON that asserts some number of claims.
- **Amazon SQS:** a fully managed message queuing service for microservices, distributed systems, and serverless applications.
- **Google Cloud Pub/Sub:** provides messaging between applications and is designed to provide reliable, many-to-many, asynchronous messaging between applications.
- **Google Calendar:** allows you to use Google Calendar API V3: get Calendars, events, synchronize with your own calendar...
- **Google FCM:** sends notifications using Firebase Cloud Messaging.

HTTP/2

HTTP/2 is an evolution of the HTTP 1.1 protocol, it basically tries to be more efficient when using networks. The semantics are the same, so it's designed to be compatible with old protocols.

HTTP 1.1 Limitations

HTTP 1.1 is limited to processing one request per connection, so usually clients use more than one connection to request files from servers. But this raises a problem, because when there are too many open TCP connections, there is a race between clients to use the server resources and performance is lower and lower as more clients connect to servers.

Main features

- HTTP/2 is a binary protocol (remember that HTTP 1.1 is a text protocol).
- HTTP/2 works over TLS and ALPN.
- It's multiplexed (allows you to send more than one request over a single TCP Connection).
- Server can push responses to clients.
- Reduces Round Trip Times, so clients can load faster.

HTTP/2 introduces other improvements, more details: RFC7540.

HTTP/2 requires our custom Indy version because it requires ALPN protocol.

Components

- **TsgcHTTP2Client:** client component that fully supports HTTP/2 protocol (sgcWebSockets 100% Pascal code, without external libraries).
- **TsgcWebSocketHTTPServer:** server component that fully supports HTTP/2 protocol (sgcWebSockets 100% Pascal code, without external libraries). By default HTTP/2 is disabled, you can enable using HTTPOptions property and set Enable = true.
- **TsgcWebSocketServer_HTTPAPI:** server component that supports HTTP/2 protocol (Microsoft implementation and Requires Windows 2016+ or Windows 10+).
- **DataSnap Servers:** datasnap server can support HTTP/2 protocol too.

APIs

- **Apple Push Notifications:** push user-facing notifications to the user's device from a server provider.

TsgcHTTP2Client

TsgcHTTP2Client implements Client HTTP/2 Component and can connect to HTTP/2 servers.

Introduction

TsgcHTTP2Client implements Client HTTP/2 Component and can connect to HTTP/2 servers. Follow the steps below to configure this component:

1. Create a new instance of **TsgcHTTP2Client** component.

2. Send the request to server and process the response using OnHTTP2Response event. example:

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2Response = OnHTTP2ResponseEvent;
oClient->Get("https://www.google.com");

void OnHTTP2ResponseEvent(TObject *Sender, const
    TsgcHTTP2ConnectionClient *Connection, const TsgcHTTP2RequestProperty *Request,
        const TsgcHTTP2ResponseProperty *Response)
{
  ShowMessage(Response->DataString);
}
```

Reference

- Properties — 19 properties available on the component.
- Methods — 19 public methods.
- Events — 13 events you can handle.

Guides

- **Requests**
  - Request HTTP/2 Method
  - HTTP/2 Server Push
  - Download File
  - HTTP/2 Partial Responses
  - HTTP/2 Headers
- **Connection**
  - Client Close Connection
  - Client Keep Connection Active
  - HTTP/2 Reason Disconnection
  - Client Pending Requests
- **Authentication**
  - Client Authentication
  - HTTP/2 and OAuth2
- **Classes**
  - TsgcHTTP2ConnectionClient
  - TsgcHTTP2RequestProperty

- TsgcHTTP2ResponseProperty

TsgcHTTP2Client | Request HTTP/2 Method

HTTP/2 Client can work in blocking and non-blocking mode, internally the component works in a secondary thread and requests are processed asynchronously, but you can call a request and wait till this request is completed.

Find below an example of how client can request an HTML page to a HTTP/2 Server and how can work in both modes.

Asynchronous Mode

Get the following url: https://www.google.com and be notified when client receives the full response. After you call **GETASYNC** method, the process continues and OnHTTP2Response event is called when response is received.

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2Response = OnHTTP2ResponseEvent;
oClient->GetAsync("https://www.goooogle.com");
void OnHTTP2ResponseEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection,
  const TsgcHTTP2RequestProperty *Request, const TsgcHTTP2ResponseProperty *Response)
{
  ShowMessage(Response->Headers->Text + #13#10 + Response->DataString);
}
```

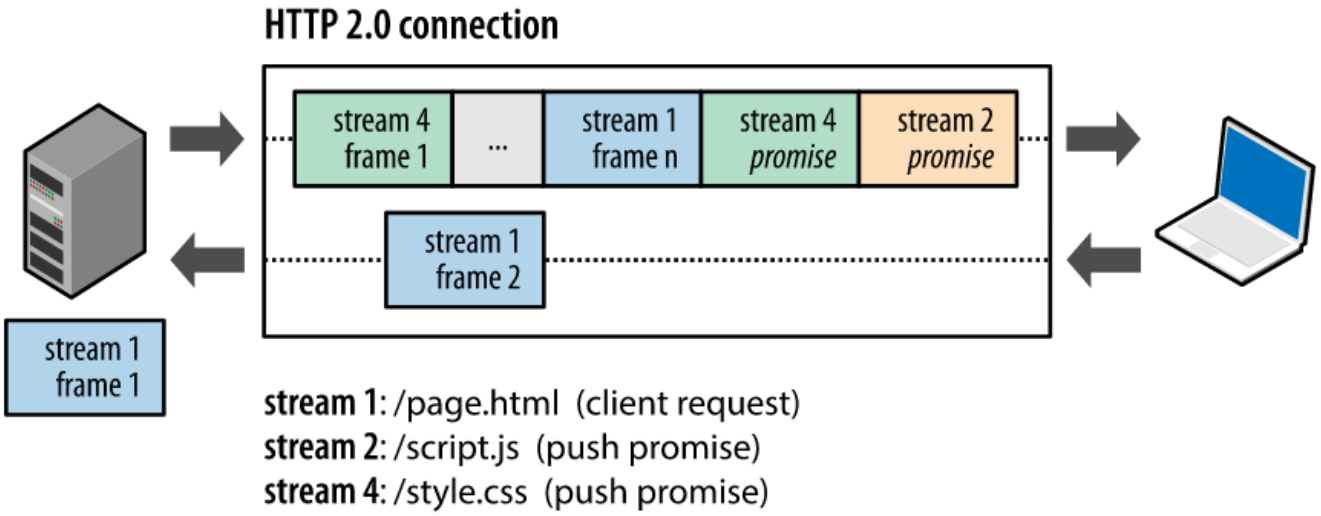
Blocking Mode

Get the following url: https://www.google.com and wait till client receives the full response. After you call **GET** method, the process waits till response is received or time out is reached.
You can access to the Raw Response data, using Response property of HTTP/2 client. Here you can access to Raw Headers, Status response code, Charset and more.

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
vResponse = oClient->Get("https://www.goooogle.com");
if (oClient->Response->Status == 200)
  ShowMessage("Response from server: " + vResponse);
else
  ShowMessage("Response Code: " + IntToStr(oClient->Response->Status));
```

Requests | HTTP/2 Server Push

Server Push is the ability of the server to send multiple responses for a single client request. That is, in addition to the response to the original request, the server can push additional resources to the client, without the client having to request each one explicitly.



Every time server sends to client a PushPromise message, OnHTTP2PushPromise event is called. When the client receives a PushPromise, it means that the server will send this resource in the next packets, so the client can accept or reject it.

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2PushPromise = OnHTTP2PushPromiseEvent;
oClient->Get("https://http2.golang.org/serverpush");
...
void OnHTTP2PushPromiseEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection,
  const TsgcHTTP2_Frame_PushPromise *PushPromise, bool &Cancel)
{
  if (PushPromise->URL == "/serverpush/static/godocs.js")
  {
    Cancel = true;
  }
  else
  {
    Cancel = false;
  }
}
```

TsgcHTTP2Client | HTTP/2 Download File

When the client requests a file from the server, use OnHTTP2Response event to load the stream response.

Large File Downloads

When downloading large files (hundreds of MB or more), set **HTTP2Options.ReadTimeout** to **0** (no timeout) to ensure the transfer completes without being interrupted by the default 60-second timeout:

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->HTTP2Options->ReadTimeout = 0; // no timeout for large files
oClient->Get("https://server/largefile", oStream);
```

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2Response = OnHTTP2ResponseEvent;
oClient->Get("https://http2.golang.org/file/gopher.png");
...
void OnHTTP2ResponseEventEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection,
  const TsgcHTTP2RequestProperty *Request, const TsgcHTTP2ResponseProperty *Response)
{
  TFileStream *oStream = new TFileStream("file", fmOpenWrite | fmCreate);
  try
  {
    oStream->CopyFrom(Response->Data, Response->Data->Size);
  }
  __finally
  {
    oStream->Free();
  }
}
```

TsgcHTTP2Client | HTTP/2 Partial Responses

Usually when you send an HTTP Request, server sends a response with the file requested, sometimes, instead of sending a single response, the server can send multiple responses like a stream, in these cases you can use **OnHTTP2ResponseFragment** event to capture these responses and show to user.

Example: send a request to https://http2.golang.org/clockstream and server will send a stream response every second.

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2ResponseFragment = OnHTTP2ResponseFragmentEvent;
oClient->Get("https://http2.golang.org/clockstream");
...
void OnHTTP2ResponseFragmentEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection,
  const TsgcHTTP2RequestProperty *Request, const TsgcHTTP2ResponseFragmentProperty *Fragment)
{
  ShowMessage(Fragment->DataString);
}
```

TsgcHTTP2Client | HTTP/2 Headers

TsgcHTTP2Client allows customizing Headers sent to server when client connects
Example: if you need to add this HTTP Header "Client: sgcWebSockets"

```
void OnHTTP2BeforeRequest(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection,
  ref TStringList *Headers)
{
  Headers->Add("Client: sgcWebSockets");
}
```

You can use Request.CustomHeaders to add your customized headers too.

TsgcHTTP2Client | Client Close Connection

Connection can be closed using Active property or using Close/Disconnect methods.

Active property

When connection is active and you set Active := False, the connection will be closed immediately without sending any message to server about the disconnection.

Disconnect

You can use Disconnect method (from TsgcHTTP2Client or TsgcHTTP2ConnectionClient) to disconnect the socket.

Close

This method, sends a message to server informing that connection will be closed and you can send optionally some info about the reason of the disconnection. It is a clean way to close an HTTP/2 connection.
Close method can be called from TsgcHTTP2Client or TsgcHTTP2ConnectionClient objects.

The following error reasons can be sent:

- no error
- protocol error
- internal error
- flow control error
- settings timeout
- stream closed
- frame size error
- refused stream
- cancel
- compression error
- connect error
- enhance your calm
- inadequate security
- required

TsgcHTTP2Client | Client Keep Connection Active

Once your client has connected to server, sometimes connection can be closed due to poor signal, connection errors... there are 2 properties which help to keep connection active.

HeartBeat

HeartBeat property allows you to **send a Ping** every X **seconds** to **maintain connection alive**. Some servers, close TCP connections if there is no data exchanged between peers. HeartBeat solves this problem, sending a ping every a specific interval. Usually this is enough to maintain a connection active.

The property HeartBeatType allows customizing how the HeartBeat works:

1. **hbtAlways:** sends a ping every x seconds defined in the Interval.
2. **hbtOnlyIfNoMsgRcvInterval:** sends a ping every x seconds only if no messages have been received during the latest x seconds defined in the Interval property.

Example: send a ping every 30 seconds

```
oClient = new TsgcHTTP2Client();
oClient->HeartBeat->Interval = 30;
oClient->HeartBeat->Enabled = true;
oClient->Active = true;
```

WatchDog

If WatchDog is enabled, when client detects a disconnection, WatchDog try to reconnect again every X seconds until connection is active again.

Example: reconnect every 10 seconds after a disconnection with unlimited attempts.

```
oClient = new TsgcHTTP2Client();
oClient->WatchDog->Interval = 10;
oClient->WatchDog->Attempts = 0;
oClient->WatchDog->Enabled = true;
oClient->Active = true;
```

TsgcHTTP2Client | HTTP/2 Reason Disconnection

HTTP/2 Server can disconnect a client for several reasons, when the server wants to inform the client of the reason why it is disconnecting, it sends a **GoAway** message to client with information about disconnection.

Use OnHTTP2GoAway event to catch the reason why server has disconnected (if client wants to close a connection, can use the method close to send the reason why is closing the connection).

TsgcHTTP2GoAwayProperty Object contains the information about disconnection

- **LastStreamId:** is the last stream processed by server.
- **ErrorCode:** integer which identifies the error code.
- **ErrorDescription:** description of the error, one of the following:
    - no error
    - protocol error
    - internal error
    - flow control error
    - settings timeout
    - stream closed
    - frame size error
    - refused stream
    - cancel
    - compression error
    - connect error
    - enhance your calm
    - inadequate security
    - required
- **AdditionalDebugData:** optional string which offers more information about disconnection.

TsgcHTTP2Client | Client Pending Requests

When client sends several requests, these are processed in a secondary thread, sometimes connection can be closed for any reason, and there are still requests pending. Use OnHTTP2PendingRequests event to handle these pending requests. This event is called when client detects a disconnection and there are still pending requests. This event has 2 parameters:

1. **Reconnect:** by default disable, if you set to true, client will reconnect automatically.
2. **Clear:** by default enabled, if you set to false, when client connects again, it will try to resend pending requests to server.

TsgcHTTP2Client | Client Authentication

HTTP/2 client supports 2 authentication types: Basic Authentication and OAuth2 Authentication.

Use **OnHTTP2Authorization** event to handle both types of authentication.

Basic Authentication

If server returns a header requesting Basic Authentication, set OnHTTP2Authorization the username and password.

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2Authorization = OnHTTP2AuthorizationEvent;
...
void OnHTTP2AuthorizationEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection, const string AuthTyp
{
  if (AuthType == "Basic")
  {
    UserName = "user";
        Password = "secret";
  }
}
```

Bearer Token

If server returns a header requesting Bearer Token Authentication, set OnHTTP2Authorization the token.

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2Authorization = OnHTTP2AuthorizationEvent;
...
void OnHTTP2AuthorizationEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection, const string AuthTyp
{
  if (AuthType == "Bearer")
  {
    aToken = "bearer token";
  }
}
```

Bearer value from Third-party

If you already know the Bearer Value, because you have obtained using another method, you can pass the Bearer value as an HTTP header using the following properties of the request, just set before calling any HTTP Request method:

```
TsgcHTTP2Client.Request.BearerAuthentication
 = true


TsgcHTTP2Client.Request.BearerToken = "< value of the token >"
```

OAuth2

Read the following article if you want to use our OAuth2 component with HTTP/2 client.

TsgcHTTP2Client | HTTP/2 and OAuth2

OAuth2 is a common authorization method used by several companies like Google. When you want to authenticate against Google servers to use any of their APIs, usually requires an Authentication using OAuth2.

sgcWebSockets supports OAuth2 under HTTP/2 client, there is a property called **Authentication.Token.OAuth** where you must assign an instance of TsgcHTTP_OAuth2.

How connect to GMail Google API

In order to connect to Google APIs, we will need to create an instance of TsgcHTTP_OAuth2 and fill the following data:

```
TsgcHTTP_OAuth1.AuthorizationServerOptions.AuthURL := 'https://accounts.google.com/o/oauth2/auth';
TsgcHTTP_OAuth1.AuthorizationServerOptions.TokenURL := 'https://accounts.google.com/o/oauth2/token';


TsgcHTTP_OAuth1.LocalServerOptions.IP := '127.0.0.1';
TsgcHTTP_OAuth1.LocalServerOptions.Port := 8080;


TsgcHTTP_OAuth1.OAuth2Options.ClientId := 'your client id';
TsgcHTTP_OAuth1.OAuth2Options.ClientSecret := 'your client secret';
```

After fill the OAuth2 client component, create a new instance of TsgcHTTP2Client and Assign the OAuth2 component to the HTTP/2 client.

```
 TsgcHTTP2Client1.Authentication.Token.OAuth := TsgcHTTP_OAuth1;
```

Finally, do a request to get a list of messages of account yourname@gmail.com

```
oStream := TStringStream.Create('');
Try
  TsgcHTTP2Client1.Get('https://gmail.googleapis.com/gmail/v1/users/yourname@gmail.com/messages', oStream);
  ShowMessage(oStream.DataString);
Finally
  oStream.Free;
End;
```

TsgcHTTP2ConnectionClient

TsgcHTTP2ConnectionClient is a wrapper of client HTTP/2 connections, you can access to this object on Client Events.

Methods

- **Ping:** sends a ping to server to maintain connection alive.
- **Close:** sends a message to server with information about why is disconnecting.
- **Disconnect:** closes the connection without sending any informational message to then server.
- **HTTP/2 Methods:**

  GET: The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

  HEAD: The HEAD method asks for a response identical to that of a GET request, but without the response body.

  POST: The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

  PUT: The PUT method replaces all current representations of the target resource with the request payload.

  DELETE: The HEAD method asks for a response identical to that of a GET request, but without the response body.

  CONNECT: The CONNECT method establishes a tunnel to the server identified by the target resource.

  OPTIONS: The OPTIONS method is used to describe the communication options for the target resource.

  TRACE: The TRACE method performs a message loop-back test along the path to the target resource.

  PATCH: The PATCH method is used to apply partial modifications to a resource.

TsgcHTTP2RequestProperty

This object is received as an argument OnHTTP2Response event, it allows to know the original request of the response send by the server.

Properties

- **Method:** identifies the HTTP/2 method (GET, POST...)
- **URL:** is the URL requested.
- **Request:** contains the fields of the request.

TsgcHTTP2ResponseProperty

This object is received as an argument OnHTTP2Response event, it allows to know the response sent by the server to the client.

Properties

- **Headers:** contains a list of raw headers received from server.
- **Data:** contains the raw body sent by the server as response to request.
- **DataString:** is the conversion to string of Data.
- **DataUTF8:** is the conversion to UTF8 string of Data.
- **PushPromise:** if assigned, contains the PushPromise object sent by the server to client (means that this response object has not been requested by client).

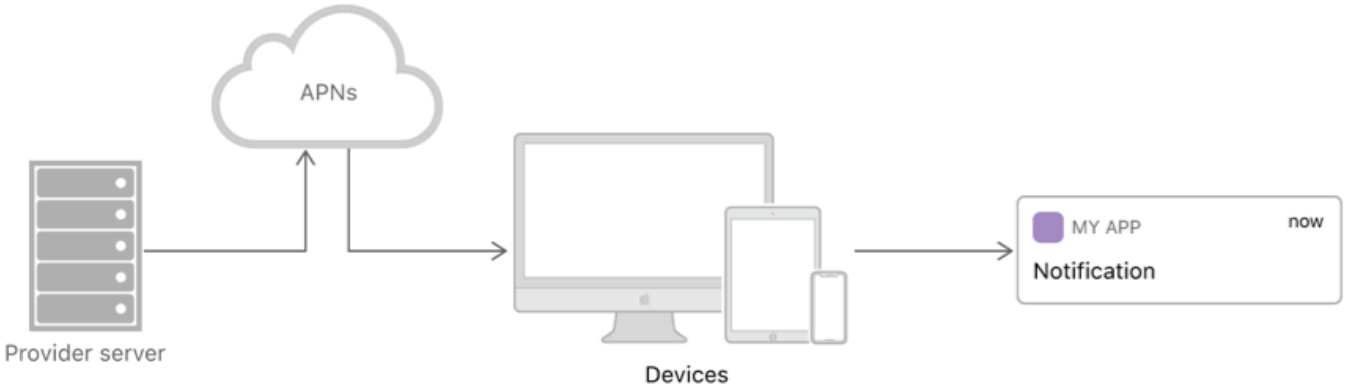
HTTP2 | Apple Push Notifications

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server

Apple allows sending push notifications to Apple devices using the Apple Push Notification Service (APNs).

When you want to send a notification to a device, the provider must send a HTTP/2 POST to APNs including the following information:

- JSON Payload with the information you want to send.
- A Device Token that identifies the user's device.
- Some HTTP Headers about how deliver the notification.
- A SSL Certificate or a JWT Token to Authenticate your request against APNs



What's required to Send Notifications

In order to send notifications to your device using Rad Studio, you must follow the next steps

- Register your APP with APNs
- Generate a Remote Notification
- Sending Notification Requests to APNs
    - Token-Based Connection to APNs
    - Certificate-Based Connection to APNs

APN | Generate a Remote Notification APNs

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/generating_a_remote_notification

The Apple Notifications use a JSON payload to send the notification object. The maximum size of the payload is 4096 bytes.

JSON Payload Samples

Simple alert message

```
{
  "aps":{
    "alert":"Alert from sgcWebSockets!"
  }
}
```

Alert with title and subtitle.

```
{
   "aps" : {
      "alert" : {
         "title" : "Game Request",
         "subtitle" : "Five Card Draw"
         "body" : "Bob wants to play poker",
      },
      "category" : "GAME_INVITATION"
   },
   "gameID" : "12345678"
}
```

APN | Sending Notification Requests to APNs

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server

Send your remote notification payload and device token information to Apple Push Notification service (APNs).

How to Connect to APNs

You must use HTTP/2 protocol and at least TLS 1.2 or later to establish a successful connection between your Server Provider and one of the following servers:

Development Server: https://api.sandbox.push.apple
Production Server: https://api.push.apple

Sample Code

Create a new instance of TsgcHTTP2Client and call the method POST to send a notification to APNs.

```
TsgcHTTP2Client *oHTTP = new TsgcHTTP2Client();
try
{
  // ... requires authorization code
  TStringStream *oStream = new TStringStream("{\"aps\":{\"alert\":\"Alert from sgcWebSockets!\"}}",
    TEncoding::UTF8);
  try
  {
    oHTTP->Post("https://api.push.apple/3/device/device_token", oStream);
    if (oHTTP->Response->Status == 200)
    {
      ShowMessage("Notification Sent Successfully");
    }
    else
    {
      ShowMessage("Notification error");
    }
  }
  __finally
  {
    oHTTP->Free();
  }
}
__finally
{
  oHTTP->Free();
}
```

To send notifications, you must establish either **token-based** or **certificate-based** trust with APNs using HTTP/2 protocol and TLS 1.2 or later.

- Token-Based Connection to APNs
- Certificate-Based Connection to APNs

APNs Trusted | Token-Based Connection to APNs

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/establishing_a_token-based_connection_to_apns

Secure your communications with Apple Push Notification service (APNs) by using stateless authentication Tokens.

First you must obtain an **Encryption Key** and a **Key ID** from Apple Developer Account. Once you complete a successful registration, you will obtain a 10-Character string with the Key ID and an Authentication Token signing key as a .p8 file extension.

You must use the sgcWebSockets JWT Client to generate a JWT using **ES256** as algorithm. The token must not be generated for every HTTP/2 request, the token must not be refreshed before 20 minutes and not after 60 minutes.

Configure JWT Client

Configure the JWT Client with the following values:

- **JWTOptions.Header.Algorithm:** is the encryption algorithm you used to encrypt the token. APNs supports only the ES256 algorithm.
- **JWTOptions.Header.kid:** is the 10-character Key ID obtained from your developer account.
- **JWTOptions.Payload.iss:** the value for which is the 10-character Team ID you use for developing your company's apps. Obtain this value from your developer account.
- **JWTOptions.Payload.iat:** The "issued at" time, whose value indicates the time at which this JSON token was generated. Specify the value as the number of seconds since Epoch, in UTC. The value must be no more than one hour from the current time.
- **JWTOptions.RefreshTokenAfter:** set the value in seconds to 40 minutes (60*40).

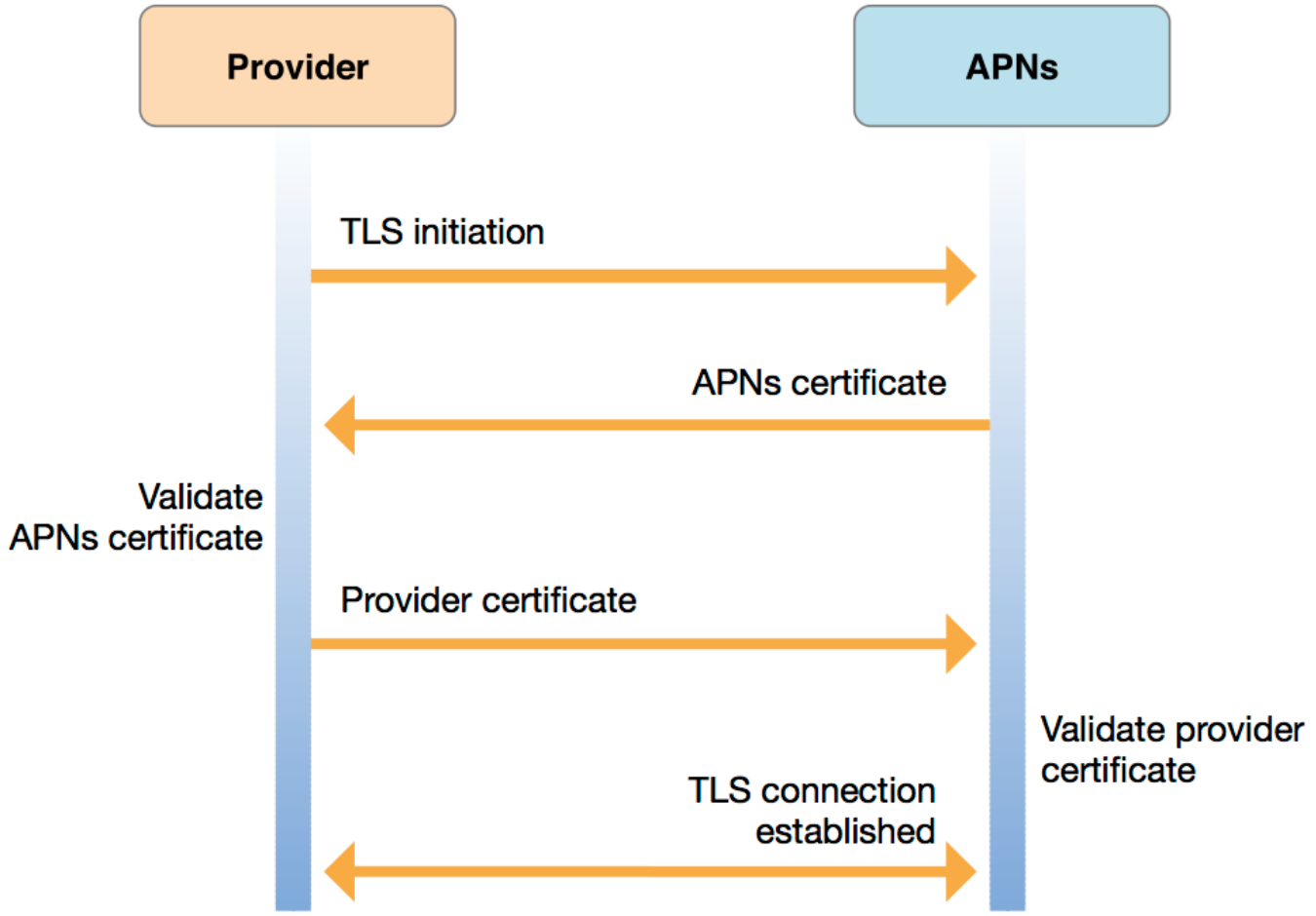
Using Token-Based connections requires sending the **apns-topic** with the value of your app's bundle ID/app id (example: com.example.application).

```
TsgcHTTP2Client *oHTTP = new TsgcHTTP2Client(NULL);
oHTTP->TLSOptions->IOHandler = iohOpenSSL;



TsgcHTTP_JWT_Client *oJWT = new TsgcHTTP_JWT_Client(NULL);
oHTTP->Authentication->Token->JWT = oJWT;
oJWT->JWTOptions->Header->alg = jwtES256;
oJWT->JWTOptions->Header->kid = "apple key id";
oJWT->JWTOptions->Payload->iss = "issuer";
oJWT->JWTOptions->Payload->iat = StrToInt64(GetDateTimeUnix(Now, False));
oJWT->JWTOptions->Algorithms->ES->PrivateKey->LoadFromFile("AuthKey_**.p8");
oJWT->JWTOptions->RefreshTokenAfter = 60*40;



oHTTP->Request->CustomHeaders->Clear();
oHTTP->Request->CustomHeaders->Add("apns-topic: com.example.application");
```

Certificate-Based Connection to APNs

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/establishing_a_certificate-based_connection_to_apns

You can secure your communications with Apple Push Notification service (APNs) using a certificate obtained from Apple.



First enter in your developer account and **create a new certificate** for Apple Push Notification service

Once you have downloaded your certificate, the sgcWebSockets HTTP/2 client allows you to use 2 security IOHandlers (only for windows, for other personalities only openSSL is supported).

- OpenSSL
- SChannel (only for windows)

OpenSSL

If you use OpenSSL, you must deploy the OpenSSL libraries with your application. Before setting the certificate with the TsgcHTTP2Client, this certificate must first be converted to PEM format because OpenSSL doesn't allow importing P12 certificates directly.

Use the following commands to convert a single P12 certificate to a certificate in PEM format and a private key file

create PEM certificate file

```
openssl pkcs12 -in INFILE.p12 -out OUTFILE.crt -nokeys
```

Create Private Key file

```
openssl pkcs12 -in INFILE.p12 -out OUTFILE.key -nodes -nocerts
```

Once you have your certificate and private key in PEM format, you can configure the TsgcHTTP2Client as follows.

```
TsgcHTTP2Client *oHTTP = new TsgcHTTP2Client(NULL);
oHTTP->TLSOptions->IOHandler = iohOpenSSL;
oHTTP->TLSOptions->CertFile = "certificate_file.pem";
oHTTP->TLSOptions->KeyFile = "private_key.pem";
oHTTP->TLSOptions->Password = "certificate password";
oHTTP->TLSOptions->Version = tls1_2;
```

SChannel

If you use SChannel there is no need to deploy any libraries and the certificate downloaded from Apple can be directly imported without the need of a previous conversion to PEM format.
Set the property UseLegacyCredentials to true when using SChannel as IOHandler.

```
TsgcHTTP2Client *oHTTP = new TsgcHTTP2Client(NULL);
oHTTP->TLSOptions->IOHandler = iohSChannel;
oHTTP->TLSOptions->SChannel_Options->UseLegacyCredentials = true;
oHTTP->TLSOptions->CertFile = "certificate_file.p12";
oHTTP->TLSOptions->Password = "certificate password";
oHTTP->TLSOptions->Version = tls1_2;
```

Errors

If you get the error **"missing topic"** most probably you are using an universal certificate (certificates that can be used for push notifications, voip...) which requires to set the topic name with the value of your app's bundle ID/app id (example: com.example.application). Just set the apns-topic header with the correct value in the Request property of the HTTP/2 client.

```
oHTTP->Request->CustomHeaders->Clear();
oHTTP->Request->CustomHeaders->Add("apns-topic: com.example.application");
```

QUIC

QUIC is the UDP-based, TLS 1.3-encrypted, multiplexed transport defined in RFC 9000, and the transport that HTTP/3 runs on.

Introduction

QUIC is a general-purpose transport protocol specified across three core documents:

RFC	Title	What it defines
RFC 9000	QUIC: A UDP-Based Multiplexed and Secure Transport	The transport itself: packets, frames, streams, connection IDs, flow control, connection lifecycle and migration.
RFC 9001	Using TLS to Secure QUIC	How TLS 1.3 is embedded in the handshake, the key schedule, packet protection and header protection.
RFC 9002	QUIC Loss Detection and Congestion Control	Acknowledgements, RTT estimation, loss detection, probe timeouts and the congestion controller.

Why QUIC exists

QUIC was designed to fix problems that are structural in TCP plus TLS, and that cannot be fixed without changing the transport itself.

- **It runs over UDP.** The transport lives in user space rather than in the operating system kernel, so it can evolve without waiting for OS and middlebox upgrades.
- **Security is not optional.** TLS 1.3 is part of the transport, not a layer bolted on top. Every QUIC connection is encrypted and authenticated, and most of the packet header is protected as well.
- **No transport head-of-line blocking.** A single connection carries many independent streams. With TCP a single lost segment stalls everything behind it, because TCP delivers one ordered byte stream. In QUIC a lost packet only stalls the streams whose data it carried, the other streams keep being delivered.
- **Faster handshake.** The transport handshake and the TLS handshake are combined into the same flight, so a fresh connection completes in one round trip instead of the separate TCP and TLS exchanges. A resumed session can carry 0-RTT application data in the very first flight.
- **Connection migration.** A connection is identified by a connection ID rather than by the four-tuple of addresses and ports, so it survives a client changing network, for example a phone moving from Wi-Fi to mobile data.

QUIC engines

This library contains two QUIC engines. They are represented by the `TsgcQUICMode` enumeration declared in `sgcSSL_QUIC.pas`:
The engine is not set directly. It follows the TLS `IOHandler` of the QUIC options (`SSLQUICOptions` on the QUIC client, `TLSOptions` on the HTTP/3 client, `QUICOptions.SSLOptions` on the servers):

- `iohOpenSSL`, the default: the engine is detected when OpenSSL is loaded. If OpenSSL exports the built-in QUIC stack (`OSSL_QUIC_client_method`, OpenSSL 3.5 or later), `qmBuiltin` is used. Otherwise, if it exports the OpenSSL 3.5 QUIC TLS API (`SSL_set_quic_tls_cbs`), the sgc engine `qmCustom` is used as the fallback. `IsOpenSSL_QUIC_Available` and `IsOpenSSL_QUIC_TLS_Available`, declared in `sgcIdSSLOpenSSLHeaders`, report these two APIs. If neither is available, for example with a quictls build, the component raises an exception saying that QUIC is not available.
- `iohNativeTLS`: `qmCustom` is always used, and the TLS 1.3 handshake runs on the native TLS engine of the sgcCrypto pack. Session tickets, resumption and 0-RTT work as on OpenSSL.

With the OpenSSL binaries shipped with the library you get `qmBuiltin`. The servers report the engine they started on in the read-only QUICMode property.
With `iohNativeTLS`, the sgc engine protects packets natively: packet and header protection (AES-128-GCM, AES-256-GCM, ChaCha20-Poly1305), key derivation, key update and the Retry integrity tag run on sgcCrypto instead of OpenSSL, per connection and without any define. Together with the native TLS 1.3 handshake, QUIC and

HTTP/3 run without OpenSSL at all, see SSL without OpenSSL. With `iohOpenSSL`, the packet protection follows `sgcCryptoBackend` (or the `SGC_NATIVE_CRYPTO` define).

Engine comparison

The table below reflects what the code actually does, not what QUIC is capable of in principle.

	`qmBuiltin`	`qmCustom`
Transport implementation	OpenSSL 3.5+ native QUIC. The library drives it through `OSSL_QUIC_client_method`, `OSSL_QUIC_server_method`, `SSL_new_listener` and `SSL_accept_connection`.	Pascal QUIC stack inside the library (the `sgcQUIC_*` units: packets, frames, congestion control, flow control), with the TLS 1.3 handshake driven through the OpenSSL 3.5 QUIC TLS API or the native TLS engine.
Selected when	`IOHandler = iohOpenSSL` and OpenSSL exports the built-in QUIC stack. This is the default with mainline OpenSSL 3.5+.	`IOHandler = iohNativeTLS`, or OpenSSL exports the QUIC TLS API but not the built-in QUIC stack.
Used by TsgcQUICClient and TsgcHTTP3Client	Yes	Yes
Used by TsgcQUICServer and TsgcHTTP3Server	Yes	Yes. An empty `Host` listens dual-stack, on IPv4 and IPv6, as on the built-in engine.
Multiple streams per connection through the component API	Yes. `OpenStream` maps onto `SSL_new_stream`, and peer-opened streams can be harvested.	Yes. `OpenStream`, `WriteStreamData` and `ConcludeStream` work with per-stream flow control and retransmission. The HTTP/3 client opens its control and QPACK streams and reuses one QUIC connection for later requests.
WebTransport	Yes. WebTransport needs multiple streams per connection, so it needs this engine.	No.

In short: `qmBuiltin` is the default engine with mainline OpenSSL 3.5+. `qmCustom` is the fallback when the built-in stack is not available, and the engine used with `iohNativeTLS`.

Streams

A QUIC connection is a container for streams. A stream is an ordered, flow-controlled sequence of bytes, and streams are independent of one another, which is what removes transport head-of-line blocking. Streams come in two shapes:

- **Bidirectional**, both endpoints can send on the stream. An HTTP/3 request and its response travel on one bidirectional stream.
- **Unidirectional**, only the endpoint that opened the stream can send. HTTP/3 uses these for its control stream, its QPACK encoder and decoder streams, and for server push.

Every stream has a 62-bit stream ID. The two least significant bits of the ID encode who opened the stream and whether it is bidirectional, which is the layout the library mirrors in `TsgcQUICStreamTypes` (`sgcQUIC_Types.pas`):

Low 2 bits	Enumeration value	Meaning
`0x00`	`qcstClientBidirectional`	Client-initiated, bidirectional
`0x01`	`qcstServerBidirectional`	Server-initiated, bidirectional
`0x02`	`qcstClientUnidirectional`	Client-initiated, unidirectional
`0x03`	`qcstServerUnidirectional`	Server-initiated, unidirectional

Stream IDs are assigned in increasing order within each of those four categories, and how many streams a peer may open is bounded by the transport parameters exchanged during the handshake. The QUIC components expose a single default stream out of the box. Additional streams are opt-in through `EnableMultiStream` and `OpenStream`, and as noted above they require the builtin engine.

Components

Component	Description
TsgcQUICClient	Low-level QUIC client. Opens a QUIC connection to a server and exchanges raw application bytes over QUIC streams. Use it when you want QUIC as a transport without HTTP semantics.
TsgcQUICServer	Low-level QUIC server. Listens for QUIC connections and exchanges raw application bytes over QUIC streams.

The HTTP/3 components build on this layer. `TsgcHTTP3Server` descends directly from `TsgcQUICServer` and adds the HTTP/3 framing, settings and request handling on top of the QUIC connections its ancestor accepts. `TsgcHTTP3Client` is a separate component rather than a descendant of `TsgcQUICClient`, but it drives the same underlying QUIC transport. See HTTP/3.

Quick start

The client connects to a host and port, then writes application bytes on the default stream. Full property, method and event documentation lives on the component pages.

```
oClient = new TsgcQUICClient(NULL);
oClient->Host = "localhost";
oClient->Port = 4433;
oClient->OnQUICConnect = OnQUICConnectHandler;
if (oClient->Connect(10000))
  oClient->WriteData("hello over QUIC");
```

The server binds a UDP port, loads a certificate and key, and then raises an event for every buffer it receives.

```
oServer = new TsgcQUICServer(NULL);
oServer->Port = 4433;
oServer->QUICOptions->SSLOptions->CertFile = "cert.pem";
oServer->QUICOptions->SSLOptions->KeyFile = "key.pem";
oServer->OnQUICData = OnQUICDataHandler;
oServer->Start();
```

QUIC always runs over UDP, so make sure the UDP port is open on any firewall in the path. A port that works for TCP is not automatically reachable over UDP.
The listener binds the address in `Host`. A value holding a colon is an IPv6 literal, and any other value, an IPv4 literal or a host name, is resolved over both families. An empty `Host` binds the dual-stack wildcard, so one listener serves IPv4 and IPv6 peers on the same port.

Requirements

Requirement	Detail
Edition	All-Access. The `SGC_PACK_HTTP3` pack that enables QUIC sits in the `SGC_EDT_ALL` block of `sgcVer.inc`.
Pack define	`SGC_PACK_HTTP3`. It ships commented out in `sgcVer.inc`, so it must be enabled explicitly before QUIC is compiled in.

Feature defines	With `SGC_PACK_HTTP3` and `SGC_INDY_LIB` both enabled, `sgcVer.inc` defines `SGC_QUIC`, `SGC_HTTP3` and `SGC_WEBTRANSPORT`. The QUIC components are compiled under `SGC_QUIC`.
Indy	`SGC_INDY_LIB` is required. The QUIC defines are nested inside an `SGC_INDY_LIB` guard, so QUIC is not available in an Indy-free build.
OpenSSL	OpenSSL 3.5 or later. The server components need the native QUIC listener (`SSL_new_listener`, `SSL_accept_connection`) that was introduced in 3.5, and raise an exception on older versions.
Units	`sgcHTTP.pas` declares the components. The transport lives in the `sgcQUIC_*` units and the OpenSSL glue in `sgcSSL_QUIC.pas`.

To enable QUIC, edit `sgcVer.inc` and uncomment the pack define inside the All-Access block:

See also

- TsgcQUICClient
- TsgcQUICServer
- HTTP/3

TsgcQUICClient

TsgcQUICClient is a low-level QUIC transport client (RFC 9000) that opens a QUIC connection to a server over UDP and exchanges raw bytes over QUIC streams.

Introduction

QUIC is the transport protocol standardised in **RFC 9000**, with its TLS integration in **RFC 9001** and its loss detection and congestion control in **RFC 9002**. It is the transport HTTP/3 runs on. Compared with TCP plus TLS it changes four things that matter to an application:

- **It runs over UDP.** QUIC lives in user space on top of UDP datagrams, so it does not depend on the operating system TCP stack and can be deployed without kernel changes. The trade-off is that a network which blocks or rate-limits UDP will block QUIC.
- **TLS 1.3 is built in, not layered on.** There is no such thing as an unencrypted QUIC connection: the handshake is part of the transport, and even most of the packet header is protected. This also collapses the round trips, a fresh connection is established in 1-RTT instead of the TCP handshake followed by a separate TLS handshake.
- **Streams are multiplexed inside one connection.** A connection carries many independent streams, each ordered and flow-controlled on its own. A packet lost on one stream does not stall the others, which removes the head-of-line blocking that HTTP/2 suffers from when multiplexing over a single TCP connection. Streams can be bidirectional or unidirectional, and either side can open them.
- **Connections are identified by a connection id, not by the 4-tuple.** This is what lets QUIC survive a change of IP address or port in the protocol design.

TsgcQUICClient exposes this transport directly, as raw bytes on streams. It does **not** speak HTTP/3 or any other application protocol: use TsgcHTTP3Client when you want HTTP semantics over QUIC. Reach for TsgcQUICClient when you are carrying your own protocol over QUIC, or testing a QUIC endpoint.

Scope worth knowing before you start. The component is a thin client over the QUIC transport, and two of its limits surprise people:

- There is **no read path until you call EnableMultiStream**. A plain client can connect and send, but nothing it receives is reported. OnQUICStreamData is the only data-received event, and it stays inert until multi-stream mode is switched on. There is no OnQUICData on the client, unlike TsgcQUICServer.
- **ALPN is fixed to `h3`** and cannot be configured. 0-RTT resumption and connection migration are implemented in the underlying transport class but are not surfaced as properties on this component.

The two QUIC engines

The library can drive QUIC through either of two engines. **The engine follows SSLQUICOptions.IOHandler**:

1. With `iohOpenSSL` (the default), if the OpenSSL build exports `OSSL_QUIC_client_method`, the **builtin engine** (`qmBuiltin`) is used. Here OpenSSL 3.5 or later owns the whole QUIC connection and the library drives it through the OpenSSL stream API.
2. Otherwise, if the build exports the OpenSSL 3.5 QUIC TLS API (`SSL_set_quic_tls_cbs`), the **sgc engine** (`qmCustom`) is used as the fallback. This is the pure Pascal QUIC stack in the `sgcQUIC_*` units: it builds and parses the packets itself and uses OpenSSL only to drive the TLS 1.3 handshake. If neither API is available, for example with a quictls build, initialising the connection raises an exception.
3. With `iohNativeTLS` the sgc engine is always used and the TLS 1.3 handshake runs on the native TLS engine, with session tickets, resumption and 0-RTT. The packet protection is native too, so OpenSSL is not loaded at all.

With the OpenSSL 3.5 binaries bundled with the demos `qmBuiltin` is selected. `IsOpenSSL_QUIC_Available` and `IsOpenSSL_QUIC_TLS_Available`, declared in `sgcIdSSLOpenSSLHeaders`, report which of the two OpenSSL APIs the loaded library exports. The engines compare as follows:

Feature	qmBuiltin (OpenSSL 3.5+)	qmCustom (sgc engine)
Connect, disconnect, send on the default stream	Yes	Yes
EnableMultiStream	Yes	Yes

OpenStream / WriteStreamData	Yes	Yes
OnQUICStreamData (the read path)	Yes, after EnableMultiStream	Yes, after EnableMulti-Stream
QUICOptions.Settings tuning	Mostly negotiated by OpenSSL, values not consulted	Yes, drives the Pascal stack

See QUIC for the full engine comparison.

Quick start

1. Create a **TsgcQUICClient** and set **Host** and **Port** of the QUIC server. QUIC runs over UDP, so Port is a UDP port, commonly 443.
2. Configure TLS through **SSLQUICOptions**. Verification is **off** by default, so turn `VerifyCertificate` on and point `RootCertFile` at your CA for anything but a throwaway test. With verification on the certificate must now also match the host you are connecting to, so a self-signed certificate, or one issued for a different name, is rejected where it used to be accepted.
3. Handle **OnQUICConnect**, **OnQUICDisconnect** and **OnQUICException**. Handling OnQUICException is not optional: connect failures do not raise, they are reported there, and Connect merely returns False.
4. Open the connection with **Connect** (blocking, returns a result) or **Start** / `Active := True` (non-blocking).
5. If you need to **receive** anything, call **EnableMultiStream** once connected and handle **OnQUICStreamData**. To send, use **WriteData** for the default stream, or **OpenStream** plus **WriteStreamData** for a specific one.

```
oClient = new TsgcQUICClient(NULL);
oClient->Host = "www.example.com";
oClient->Port = 443;

oClient->SSLQUICOptions->VerifyCertificate = true;
oClient->SSLQUICOptions->RootCertFile = "ca.pem";
oClient->SSLQUICOptions->ConnectTimeout = 10000;

oClient->OnQUICConnect = OnQUICConnectEvent;
oClient->OnQUICStreamData = OnQUICStreamDataEvent;
oClient->OnQUICException = OnQUICExceptionEvent;

if (oClient->Connect())
  oClient->WriteData("Hello QUIC");
```

Reference

- Properties — 10 properties available on the component.
- Methods — 10 public methods.
- Events — 4 events you can handle.

Key Properties

Name	Description
Active	Opens or closes the QUIC connection to the remote server.
Host	IP address or DNS name of the QUIC server the client will connect to.
Port	UDP port used to connect to the QUIC server.
SSLQUICOptions	TLS 1.3 settings: certificates, peer verification and the handshake timeout. Verification is off by default.
QUICOptions	Transport tuning for the QUIC connection, such as PMTU and flow control windows.

Key Methods

Name	Description
Connect	Opens the QUIC connection and waits until it is established or the timeout expires.
Disconnect	Closes the QUIC connection to the server.
WriteData	Sends data to the server over the default QUIC stream.
EnableMultiStream	Switches the live connection into multi-stream mode. Required before anything can be received.
OpenStream	Opens a new client-initiated QUIC stream and returns it, or nil on failure.

Key Events

Name	Description
OnQUICConnect	Fires once the QUIC connection has been established and the client is ready to send data.
OnQUICStreamData	Fires with the bytes read from a QUIC stream. Inert until EnableMultiStream has been called.
OnQUICDisconnect	Fires when the QUIC connection has been closed, reporting the close code and reason.
OnQUICException	Fires when an exception is raised on the QUIC connection. The only way to learn why a connect failed.

Complete Example

Connect to a QUIC server, enable multi-stream so data can be received, send a message on a dedicated stream and log everything that comes back.

```
#include "sgcHTTP.hpp"
#include "sgcQUIC_Client.hpp"
#include "sgcQUIC_Classes.hpp"

void __fastcall TForm1::Start()
{
  FClient = new TsgcQUICClient(NULL);
  FClient->Host = "www.example.com";
  FClient->Port = 443;

  // TLS 1.3. Verification is off by default, turn it on.
  FClient->SSLQUICOptions->VerifyCertificate = true;
  FClient->SSLQUICOptions->RootCertFile = "ca.pem";
  // the handshake timeout, 10000 ms is already the default
  FClient->SSLQUICOptions->ConnectTimeout = 10000;

  FClient->OnQUICConnect = OnQUICConnectEvent;
  FClient->OnQUICStreamData = OnQUICStreamDataEvent;
  FClient->OnQUICDisconnect = OnQUICDisconnectEvent;
  FClient->OnQUICException = OnQUICExceptionEvent;

  if (!FClient->Connect(10000))
    Log("connect failed");
}

void __fastcall TForm1::OnQUICConnectEvent(TObject *Sender, TsgcQUICConnection *Connection)
{
  Log("connected");

  FClient->EnableMultiStream();
```

```
  TsgcQUICClientStream *oStream = FClient->OpenStream();
  if (oStream)
    FClient->WriteStreamData(oStream, TEncoding::UTF8->GetBytes("Hello QUIC"));
  else
    Log("could not open a stream");
}

void __fastcall TForm1::OnQUICStreamDataEvent(TObject *Sender, __int64 aStreamId,
  bool aUnidirectional, const System::DynamicArray<System::Byte> aData)
{
  Log("stream " + IntToStr(aStreamId) + ": " + IntToStr(aData.Length) + " bytes");
}

void __fastcall TForm1::OnQUICDisconnectEvent(TObject *Sender, TsgcQUICConnection *Connection,
  __int64 Code, const UnicodeString Reason)
{
  if (Code == 0)
    Log("closed cleanly");
  else
    Log("closed, code " + IntToStr(Code) + ": " + Reason);
}

void __fastcall TForm1::OnQUICExceptionEvent(TObject *Sender, TsgcQUICConnection *Connection,
  Exception *E)
{
  Log("error: " + E->Message);
}
```

Requirements

- Available in the **All-Access** edition.
- **QUIC ships disabled.** Open **sgcVer.inc** and uncomment `{$DEFINE SGC_PACK_HTTP3}` inside the `{$IFDEF SGC_EDT_ALL}` block, it is delivered as `{.$DEFINE SGC_PACK_HTTP3}`. Enabling it defines **SGC_QUIC**, **SGC_HTTP3** and **SGC_WEBTRANSPORT**, then rebuild the packages.
- Those defines are nested inside `{$IFDEF SGC_INDY_LIB}`, so the **SGC_INDY_LIB** build is required.
- Add **sgcHTTP** to the uses clause for the `TsgcQUICClient` component itself, **sgcQUIC_Client** for `TsgcQUIC-ClientStream`, and **sgcQUIC_Classes** for `TsgcQUICConnection` in the event handlers.
- Requires **OpenSSL 3.5 or later** (`libssl-3` and `libcrypto-3`) for the builtin QUIC engine, which is the engine that ships. Earlier OpenSSL 3.x builds have no QUIC support and raise an exception when the connection is initialised.
- In .NET, QUIC requires **.NET 8.0 or later** and MsQuic at runtime. On netstandard2.0 the API compiles but every operation throws `PlatformNotSupportedException`.

TsgcQUICServer

TsgcQUICServer is a low-level QUIC server: it listens for QUIC connections over UDP and exchanges raw application bytes over QUIC streams.

Introduction

TsgcQUICServer implements the server side of QUIC (RFC 9000, with TLS 1.3 per RFC 9001 and loss recovery per RFC 9002). QUIC is a connection-oriented, multiplexed, always-encrypted transport that runs over UDP. This component exposes it as a **byte transport**: it hands you the bytes a peer sends and lets you write bytes back. It does not implement any application protocol on top. For HTTP/3, use TsgcHTTP3Server, which descends from this component.
What the component gives you:

- **A UDP listener.** QUIC has no TCP socket. Start creates a UDP socket, binds it, and runs a background thread that drives the QUIC event loop: accepting connections, advancing handshakes and reading streams. Remember to open the **UDP** port in the firewall, UDP and TCP are filtered independently.
- **Mandatory TLS 1.3.** QUIC has no plaintext mode, so a certificate and a private key are always required. Configure them through QUICOptions.SSLOptions. Without them, `Start` raises.
- **ALPN.** QUIC mandates ALPN, so the server negotiates an application protocol during the handshake. The built-in selection callback prefers `h3` and otherwise accepts the peer's first offer, and it is not configurable. A peer that offers nothing is rejected.
- **Streams and multiplexing.** One connection carries many independent, ordered streams, so a slow stream does not block the others (no head-of-line blocking). By default the server keeps things simple and uses the first stream the peer opens as the connection's single default data stream. Set AlwaysMultiStream to work with all of them.
- **A per-connection model.** Every accepted peer is a TsgcQUICServerConnection, which is passed to every event and is the handle you write back on.

The QUIC server runs on the OpenSSL 3.5+ built-in QUIC engine by default. It calls `OSSL_QUIC_server_method` and `SSL_new_listener` when the loaded OpenSSL library exports them. Otherwise it falls back to the sgc QUIC engine, and with `QUICOptions.SSLOptions.IOHandler = iohNativeTLS` it always uses the sgc engine with the TLS 1.3 handshake on the native TLS engine. The read-only QUICMode property reports the engine selected at start. See QUIC for the engine comparison.
Follow these steps to configure the component:
1. Create a new instance of **TsgcQUICServer**.
2. Set the listening **Port**. It defaults to `80`, inherited from the UDP base class, so always set it (`4433` and `443` are the usual QUIC choices).
3. Set the certificate and private key on **QUICOptions.SSLOptions**. Both are PEM files and both are mandatory. For a test certificate:

```
openssl req -x509 -newkey rsa:2048 -keyout server.key -out server.crt -days 365 -nodes -subj "/CN=localhost"
```

4. Handle **OnQUICConnect** and **OnQUICData**, and call **Start** inside a `try..except`, because configuration errors surface there.

```
oServer = new TsgcQUICServer(NULL);
oServer->Port = 4433;
oServer->QUICOptions->SSLOptions->CertFile = "server.crt";
oServer->QUICOptions->SSLOptions->KeyFile = "server.key";
oServer->OnQUICConnect = OnQUICConnectEvent;
oServer->OnQUICData = OnQUICDataEvent;
oServer->Start();

void __fastcall TfrmServer::OnQUICDataEvent(TObject *Sender,
  const TsgcQUICServerConnection *aConnection, const TBytes aData)
{
  // echo the received bytes back to the peer
  oServer->WriteData(aConnection, aData);
}
```

Threading. Every event is raised on the server's background listener thread, and that single thread drives the event loop for all connections. So keep handlers short, and marshal to the main thread before touching the UI.

Lifecycle. Use Start and Stop, or the Active property, which they are implemented on top of and which reports the real listener state. OnStartup and OnShutdown are raised around those transitions, so you can act from a handler or after the call returns, whichever suits you.

Relationship with TsgcHTTP3Server

`TsgcHTTP3Server_Base` descends from `TsgcQUICServer_Base`, so TsgcHTTP3Server **is** a QUIC server with the HTTP/3 protocol layered on it. Use `TsgcQUICServer` when you want a raw byte transport with your own protocol on top, and `TsgcHTTP3Server` when you want to serve HTTP/3.
If you use the HTTP/3 server, note that it claims part of this component's surface: its constructor sets AlwaysMultiStream to `True` (HTTP/3 needs the client's unidirectional control and QPACK streams, RFC 9114) and wires `OnQUICConnect`, `OnQUICData`, `OnQUICStreamData` and `OnQUICDisconnect` to its own handlers. Assigning those events, or clearing `AlwaysMultiStream`, breaks the HTTP/3 layer. `OnQUICException` is the one exception, the HTTP/3 server only claims it if you have not assigned it yourself.

Reference

- Properties — 7 properties available on the component.
- Methods — 6 public methods.
- Events — 7 events you can handle.

Two classes complete the API:

- TsgcQUICServerConnection — a single accepted connection, the object handed to every event.
- TsgcQUICServerStream — one additional QUIC stream on a connection.

Key Properties

Name	Description
QUICOptions	The whole server setup: the mandatory TLS certificate and private key, and the connection limit.
Port	UDP port the server binds to. Defaults to `80`, so always set it.
AlwaysMultiStream	Accept and read every stream the peer opens, not just the first one.
Connections	The list of QUIC connections currently accepted by the server.

Key Methods

Name	Description
Start	Loads OpenSSL, creates the QUIC listener and begins accepting connections.
Stop	Stops listening, closes every active connection and releases the QUIC listener.
WriteData	Sends application bytes to a connected peer over its default QUIC stream.
OpenStream	Opens a new server-initiated QUIC stream on a connection.

Key Events

Name	Description
OnQUICConnect	Fires when a new QUIC connection has been accepted and its first stream is ready.
OnQUICData	Fires when application data is received from a peer over its default QUIC stream.
OnQUICStreamData	Fires with bytes read from an additional QUIC stream, tagged with its stream id.
OnQUICDisconnect	Fires when a QUIC connection has been closed.

Complete Example

An echo server that reports connections and errors. Note that `PeerIP` is not populated by the Delphi server, because OpenSSL exposes no getter for it, so the session is identified by the connection instance.

```
#include "sgcQUIC_Server.hpp"
#include "sgcHTTP.hpp"

void __fastcall TfrmServer::StartServer()
{
  oServer = new TsgcQUICServer(NULL);
  oServer->Port = 4433;

  // mandatory: QUIC always runs over TLS 1.3
  oServer->QUICOptions->SSLOptions->CertFile = "server.crt";
  oServer->QUICOptions->SSLOptions->KeyFile = "server.key";
  oServer->QUICOptions->MaxConnections = 1000;

  oServer->OnQUICConnect = OnQUICConnectEvent;
  oServer->OnQUICData = OnQUICDataEvent;
  oServer->OnQUICDisconnect = OnQUICDisconnectEvent;
  oServer->OnQUICException = OnQUICExceptionEvent;
  oServer->OnStartup = OnStartupEvent;

  // Start raises on a bad configuration, OnStartup fires once it is listening
  try
  {
    oServer->Start();
  }
  catch (Exception &E)
  {
    Log("start failed: " + E.Message);
  }
}

void __fastcall TfrmServer::StopServer()
{
  oServer->Stop(); // OnQUICDisconnect is NOT raised for these connections
  Log("QUIC server stopped");
  delete oServer;
}

void __fastcall TfrmServer::OnStartupEvent(TObject *Sender)
{
  // raised once the listener is bound and accepting
  Log("QUIC server listening on UDP port " + IntToStr(oServer->Port));
}

void __fastcall TfrmServer::OnQUICConnectEvent(TObject *Sender,
  const TsgcQUICServerConnection *aConnection)
{
  Log("connected at " + DateTimeToStr(aConnection->CreatedAt) +
    " (" + IntToStr(oServer->ConnectionCount()) + " live)");
}

void __fastcall TfrmServer::OnQUICDataEvent(TObject *Sender,
  const TsgcQUICServerConnection *aConnection, const TBytes aData)
{
  // aData is a chunk, not a message: frame at the application level
  oServer->WriteData(aConnection, aData);
}

void __fastcall TfrmServer::OnQUICDisconnectEvent(TObject *Sender,
  const TsgcQUICServerConnection *aConnection, __int64 aCode,
```

```
  const UnicodeString aReason)
{
  Log("disconnected (code=" + IntToStr(aCode) + " " + aReason + ")");
}

void __fastcall TfrmServer::OnQUICExceptionEvent(TObject *Sender,
  const TsgcQUICServerConnection *aConnection, Exception *aException)
{
  // aConnection is NULL for a listener error, otherwise it is the failing session
  if (aConnection)
    Log("connection error: " + aException->Message);
  else
    Log("listener error: " + aException->Message);
}
```

Requirements

Require-ment	Detail
Edition	**All-Access** edition.
Compiler define	QUIC ships **disabled**. Uncomment `{$DEFINE SGC_PACK_HTTP3}` in the All-Access block of `sgcVer.inc`, it is delivered as `{.$DEFINE SGC_PACK_HTTP3}`. That block defines `SGC_QUIC`, `SGC_HTTP3` and `SGC_WEBTRANSPORT`, but only when `SGC_INDY_LIB` is also defined.
Unit	`sgcQUIC_Server` for the classes, `sgcHTTP` for the registered `TsgcQUICServer` component.
OpenSSL	**OpenSSL 3.5 or later** with `IOHandler = iohOpenSSL`. The built-in engine uses the native QUIC listener API (`OSSL_QUIC_server_method`, `SSL_new_listener`) and the sgc engine uses the OpenSSL 3.5 QUIC TLS API, neither of which exists in earlier versions. With `iohNativeTLS` OpenSSL is not needed. `QUICOptions.SSLOptions.OpenSSL_Options.APIVersion` must stay `oslAPI_3_0`.
TLS	A certificate and a matching private key, both PEM, are mandatory. An encrypted private key is supported, set its passphrase in `QUICOptions.SSLOptions.Password`.
.NET	net8.0 or later, and MsQuic at runtime. The QUIC server exists on every target framework for parity, but on netstandard2.0 `Start()` throws `PlatformNotSupportedException`.

HTTP/3

HTTP/3 is the third major version of HTTP, defined in RFC 9114. It keeps HTTP semantics unchanged and replaces the transport with QUIC.

Introduction

HTTP/3 keeps everything you already know about HTTP: the same methods, the same status codes, the same header fields, the same caching and authentication rules. What changes is what carries them. HTTP/1.1 and HTTP/2 run over TCP with TLS layered on top. HTTP/3 runs over QUIC, a UDP-based transport with TLS 1.3 built in.
Because HTTP semantics are unchanged, an application ported from HTTP/1.1 or HTTP/2 to HTTP/3 behaves the same way. You issue a GET, you get a status code and a body back.

RFC	Title	What it defines
RFC 9114	HTTP/3	The mapping of HTTP onto QUIC: frames, request and response streams, unidirectional stream types, settings, server push and the HTTP/3 error codes.
RFC 9204	QPACK: Field Compression for HTTP/3	Header compression designed for QUIC, using a static table plus a dynamic table carried on dedicated encoder and decoder streams.
RFC 9000	QUIC transport	The transport HTTP/3 is carried on. See QUIC.

Why not HPACK

HTTP/2 compresses headers with HPACK, which assumes headers arrive in the order they were sent, an assumption TCP guarantees. QUIC streams can be delivered out of order, so HPACK would reintroduce head-of-line blocking at the compression layer. QPACK solves this by carrying dynamic-table updates on separate unidirectional streams and by letting a request block until only the table entries it actually depends on have arrived.

How HTTP/3 compares to HTTP/1.1 and HTTP/2

	HTTP/1.1	HTTP/2	HTTP/3
Specification	RFC 9112	RFC 9113	RFC 9114
Transport	TCP	TCP	QUIC over UDP
Encryption	Optional, TLS layered on top	Optional in the spec, TLS in practice	Always on, TLS 1.3 is part of the transport
Multiplexing	None. One request in flight per connection, so clients open several connections	Many streams over one TCP connection	Many streams, provided by the transport itself
Head-of-line blocking	Yes, at the request level	Removed at the application layer, but a lost TCP segment still stalls every stream	Removed. A lost packet only stalls the streams whose data it carried
Header compression	None, headers are plain text	HPACK (RFC 7541)	QPACK (RFC 9204)

Message format	Text	Binary frames	Binary frames over QUIC streams
Hand-shake cost	TCP handshake, then a sepa-rate TLS handshake	TCP handshake, then a sepa-rate TLS handshake	Transport and TLS com-bined in one flight, with 0-RTT on a resumed session
Connec-tion migra-tion	No, the connection is bound to the four-tuple	No	Yes, the connection is iden-tified by a connection ID
Server push	No	Yes, deprecated in practice	Yes, driven by push IDs and a client-controlled limit

Supported HTTP methods

The client supports the ten methods declared in the `Th3Methods` enumeration (`sgcHTTP3_Types.pas`). Each has a blocking method, and most also have an asynchronous variant. This table is an orientation map, see TsgcHTTP3Client for the full signatures, overloads and default parameter values.

Method	Enumeration value	Client method	Async variant
`GET`	`h3meGet`	`Get`	`GetAsync`
`HEAD`	`h3meHead`	`Head`	—
`POST`	`h3mePost`	`Post`	`PostAsync`
`PUT`	`h3mePut`	`Put`	`PutAsync`
`DELETE`	`h3meDelete`	`Delete`	`DeleteAsync`
`PATCH`	`h3mePatch`	`Patch`	`PatchAsync`
`OPTIONS`	`h3meOptions`	`RequestOptions`	`OptionsAsync`
`TRACE`	`h3meTrace`	`Trace`	`TraceAsync`
`CONNECT`	`h3meConnect`	`Connect`	`ConnectAsync`
`QUERY`	`h3meQuery`	`Query`	`QueryAsync`

Notes on two of them:

- `QUERY` is the IETF QUERY method, a work-in-progress specification. It is a safe, idempotent method that carries a request body, so it covers the common case of a read-only query too large or too structured to fit in a URL, which today is usually forced onto `POST`. It is not yet a published RFC.
- `HEAD` returns the response object rather than a string, because a `HEAD` response carries headers and no body. It has no asynchronous variant.

Discovery with Alt-Svc

A client cannot assume a server speaks HTTP/3, and there is no HTTP/3 upgrade handshake equivalent to the HTTP/2 one. The usual discovery route is the `Alt-Svc` response header (RFC 7838): a server reached over HTTP/1.1 or HTTP/2 advertises that the same origin is also available over HTTP/3 at a given host and port, and the client remembers that for future requests. The client parses this header and surfaces it through the `OnAltSvc` event, so your application can decide when to switch.

Server push

A server can push a response the client did not ask for, so a client fetching a page does not need a second round trip for a resource the server knows it will need. Each push carries a push ID. The client keeps control: it advertises how many pushes it will accept with `SetMaxPushId`, is notified of an incoming push through `OnPushPromise`, and can refuse one with `CancelPush`.

Extended CONNECT

Extended CONNECT (RFC 9220 for HTTP/3, which carries over the mechanism RFC 8441 defined for HTTP/2) turns a request stream into a tunnel for another protocol. It adds a `:protocol` pseudo-header to the `CONNECT` method, so the endpoint knows which protocol it is being asked to tunnel.
It is only available when the server advertises `SETTINGS_ENABLE_CONNECT_PROTOCOL` in its settings. The client exposes `ExtendedConnect`, which returns `False` when the server never advertised it, or when the response status is not a 2xx. Once the tunnel is open you write to it with `TunnelSend` and receive through `OnTunnelData`.

WebTransport

WebTransport is built on Extended CONNECT. It negotiates a session over an Extended CONNECT stream and then gives the application QUIC streams directly, which is what you want for low-latency, bidirectional, non-request/response traffic. The specification also offers datagrams, but they cannot be carried on the QUIC engine this library ships with, so WebTransport here is streams only. It needs both `SETTINGS_ENABLE_CONNECT_PROTOCOL` and a non-zero `SETTINGS_WEBTRANSPORT_MAX_SESSIONS` from the server, which the client checks through `ServerSupportsWebTransport`. Because a session needs several streams on one QUIC connection, WebTransport requires the builtin OpenSSL QUIC engine, see QUIC. Details are on the WebTransport page.

Components

Component	Description
TsgcHTTP3Client	Sends HTTP requests over HTTP/3, with blocking and asynchronous methods, Alt-Svc detection, server push, Extended CONNECT tunnels and WebTransport.
TsgcHTTP3Server	Accepts HTTP/3 connections and serves requests over QUIC. It descends from Tsgc-QUICServer and adds HTTP/3 framing, settings and request dispatch.
WebTransport	Sessions and streams over HTTP/3, negotiated with Extended CONNECT. Datagrams are not carried on the shipping QUIC engine.

Quick start

Connect to a host and port, then issue requests. The default port is 443.

```
oClient = new TsgcHTTP3Client(NULL);
oClient->OnResponse = OnResponseHandler;
oClient->OnAltSvc = OnAltSvcHandler;
oClient->Connect("www.example.com", 443);
vBody = oClient->Get("https://www.example.com/");
oClient->Disconnect();
```

The server binds a UDP port, loads a certificate and key, and raises an event for every request.

```
oServer = new TsgcHTTP3Server(NULL);
oServer->Port = 443;
oServer->QUICOptions->SSLOptions->CertFile = "cert.pem";
oServer->QUICOptions->SSLOptions->KeyFile = "key.pem";
oServer->OnH3Request = OnH3RequestHandler;
oServer->Start();
```

HTTP/3 travels over UDP, so port 443 must be reachable over UDP and not only over TCP. This is the most common reason a working HTTP/2 deployment fails to serve HTTP/3.
The server binds the address in `Host`. A value holding a colon is an IPv6 literal, and any other value, an IPv4 literal or a host name, is resolved over both families. An empty `Host` binds the dual-stack wildcard, so one listener serves IPv4 and IPv6 peers on the same port. The client is URL driven, so an IPv6 literal goes in brackets there, for example `oClient.Get('https://[::1]:443/')`.

Requirements

Require-ment	Detail
Edition	All-Access. The `SGC_PACK_HTTP3` pack sits in the `SGC_EDT_ALL` block of `sgcVer.inc` .
Pack de-fine	`SGC_PACK_HTTP3` . It ships commented out in `sgcVer.inc` , so it must be enabled explicitly before HTTP/3 is compiled in.
Feature de-fines	With `SGC_PACK_HTTP3` and `SGC_INDY_LIB` both enabled, `sgcVer.inc` defines `SGC_QUIC` , `SGC_HTTP3` and `SGC_WEBTRANSPORT` . The HTTP/3 components are compiled under `SGC_HTTP3` , and the WebTransport members under `SGC_WEBTRANSPORT` .
Indy	`SGC_INDY_LIB` is required. The HTTP/3 defines are nested inside an `SGC_INDY_LIB` guard.
OpenSSL	OpenSSL 3.5 or later, which provides the native QUIC support the components are built on by default. WebTransport additionally requires the builtin OpenSSL QUIC engine. With the TLS `IOHandler` set to `iohNativeTLS` , HTTP/3 runs without OpenSSL on the sgc QUIC en-gine, see SSL without OpenSSL.
Units	`sgcHTTP.pas` declares the components. HTTP/3 itself lives in the `sgcHTTP3_*` units, Web-Transport in `sgcWebTransport.pas` , and the transport in the `sgcQUIC_*` units.

To enable HTTP/3, edit `sgcVer.inc` and uncomment the pack define inside the All-Access block:

See also

- TsgcHTTP3Client
- TsgcHTTP3Server
- WebTransport
- QUIC

TsgcHTTP3Client

TsgcHTTP3Client is an HTTP/3 client (RFC 9114) that sends requests over QUIC, supporting the ten HTTP verbs synchronously and asynchronously, server push, Extended CONNECT tunnels and WebTransport.

Introduction

HTTP/3 (RFC 9114) is the third major revision of HTTP. It keeps the semantics you already know, the same verbs, status codes and headers, but replaces the transport underneath. Instead of TCP it runs over **QUIC** (RFC 9000), a UDP-based transport with TLS 1.3 built into the handshake.
That change buys three things. Encryption is **mandatory**, so there is no plaintext HTTP/3 and no TLS version to negotiate. Streams are independent, so a lost packet on one request no longer stalls every other request sharing the connection, which is the head-of-line blocking that limits HTTP/2. And the handshake is shorter, so a connection is usable sooner.
Two practical consequences are worth knowing before you start. HTTP/3 travels over **UDP**, so a firewall that only opens TCP 443 blocks it even though the port number is the same as HTTPS. And headers are compressed with **QPACK** (RFC 9204) rather than HPACK, which the component handles for you and reports on through QPACKDynamicInserts.
HTTP Verbs
The client implements the ten verbs below. The list is exhaustive: it is the `Th3Methods` enumeration the component encodes into the `:method` pseudo-header of every request.

Verb	Synchronous	Asynchronous	Stream overload	Default aContentType	Typical use
`GET`	Get	GetAsync	`Get(aURL, aResponseContent)`	—	Retrieve a resource. No request body.
`POST`	Post	PostAsync	`Post(aURL, aSource, aResponseContent)`	`application/x-www-form-urlencoded`	Submit data, typically to create a record or invoke an action.
`PUT`	Put	PutAsync	`Put(aURL, aSource, aResponseContent)`	`application/json`	Replace the target resource. Idempotent.
`DELETE`	Delete	DeleteAsync	`Delete(aURL, aResponseContent)`	—	Remove the target resource. No request body.
`HEAD`	Head	—	—	—	Read status and headers only, no body. Returns `TsgcHTTP3Response`.
`PATCH`	Patch	PatchAsync	`Patch(aURL, aSource, aResponseContent)`	`application/json`	Apply a partial update to the target resource.
`OPTIONS`	RequestOptions	OptionsAsync	`RequestOptions(aURL, aResponseContent)`	—	Inspect the methods and capabilities a resource supports.
`TRACE`	Trace	TraceAsync	`Trace(aURL, aResponseContent)`	—	Loop-back diagnostic. Often disabled by servers.

`CONNECT`	Connect	ConnectAsync	—	—	Request a tunnel. See ExtendedConnect for a tunnel that stays open.
`QUERY`	Query	QueryAsync	`Query(aURL, aSource, aResponseContent)`	`application/x-www-form-urlencoded`	Safe, idempotent request that carries a body. Suits search and filter payloads.

Two naming points are worth knowing. **OPTIONS** is sent by the method named RequestOptions, because `Options` collides with an existing identifier in Delphi and C++Builder; its asynchronous counterpart keeps the protocol name and is OptionsAsync. **CONNECT** is sent by the `Connect(aURL)` overload, which is a different method from the `Connect(aHost, aPort)` overload that opens the QUIC connection.
HEAD is the exception to the pattern: it returns a `TsgcHTTP3Response` rather than a string, and it has neither an asynchronous variant nor a stream overload. Every other verb except CONNECT also has a **stream overload** that reads the request body from a stream and writes the response body into one, which avoids building large strings in memory. Stream overloads always send `content-type: application/octet-stream`.
Synchronous and Asynchronous requests
Every verb can be sent two ways. A **synchronous** call blocks the calling thread until the response is complete and returns the body directly, bounded by ReadTimeout. An **asynchronous** call returns immediately, runs the request on a background thread and delivers the reply on OnResponse.
When you use the asynchronous methods, always handle OnError: a background request has no caller to raise to, so without that handler a failure is silent. Remember also that the events fire on the background worker thread, so synchronise before touching VCL or FMX controls.
Follow the steps below to send a synchronous request:
1. Create a new instance of **TsgcHTTP3Client**. There is no host to assign: Host and Port are read-only and report the current connection.
2. Call a verb method with an absolute URL. The connection is opened on demand from that URL, so calling Connect first is optional. Call Disconnect when finished. example:

```
TsgcHTTP3Client *oClient = new TsgcHTTP3Client(NULL);
try
{
  UnicodeString vResponse = oClient->Get("https://www.example.com/");
  ShowMessage(vResponse);
  oClient->Disconnect();
}
__finally
{
  delete oClient;
}
```

To send the request asynchronously, assign OnResponse and call GetAsync; the call returns immediately and the response is delivered on the event. Decode the body with `GetDataAsString`, which honours the charset in the `content-type` header. example:

```
TsgcHTTP3Client *oClient = new TsgcHTTP3Client(NULL);
oClient->OnResponse = OnResponseEvent;
oClient->OnError = OnErrorEvent;
oClient->GetAsync("https://www.example.com/");

void __fastcall TfrmMain::OnResponseEvent(TObject *Sender,
  TsgcHTTP3Response *aResponse)
```

```
{
  // fires on the background thread
  Log(aResponse->GetDataAsString());
}
```

Server Push
Server push lets the server send a resource the client never asked for, expecting it will be needed (RFC 9114 Section 4.6). Push is **disabled** until the client opts in by calling SetMaxPushId, which announces the highest push ID it will accept.
Each push then arrives in two stages. OnPushPromise fires first with the request headers the push will answer, which is the moment to refuse it with CancelPush when the resource is already cached. If it is not refused, the complete pushed response arrives on OnPush with the same push ID.
Extended CONNECT and WebTransport
Extended CONNECT (RFC 9220, the HTTP/3 counterpart of RFC 8441) turns a request stream into a bidirectional tunnel for another protocol. Call ExtendedConnect with the protocol token; it returns True on a 2xx status and keeps the stream open. Write to the tunnel with TunnelSend, and read from it by calling TunnelPump, which surfaces incoming bytes on OnTunnelData. The server must advertise `SETTINGS_ENABLE_CONNECT_PROTOCOL`, otherwise the call returns False.
WebTransport is that mechanism with the `webtransport` protocol token and a session object on top. WebTransportConnect performs the handshake and returns a `TsgcWebTransportSession`, and ServerSupportsWebTransport tells you in advance whether the server supports it. Peer-opened streams are only harvested while WebTransportProcessEvents is being called.
WebTransport sessions carry **streams only**. Datagrams require the RFC 9221 QUIC DATAGRAM extension, which the OpenSSL 3.5 builtin QUIC engine that ships with the library does not provide, so sending a datagram raises `TsgcQUICException` rather than lose the payload quietly.

Reference

- Properties — 13 properties available on the component.
- Methods — 31 public methods, including the 10 HTTP verbs.
- Events — 11 events you can handle.

Key Properties

Name	Description
ConnectTimeout	Maximum time in milliseconds the client waits for the QUIC connection to be established. Default 10000.
ReadTimeout	Maximum time in milliseconds the client waits for a response to arrive. Default 30000.
MaxResponseSize	Maximum response body bytes accepted per request. Default 0, meaning unlimited.
TLSOptions	TLS 1.3 parameters (certificates, verification, OpenSSL) used by the QUIC connection.
Settings	HTTP/3 SETTINGS parameters announced to the server when the connection is established.
PeerSettings	HTTP/3 SETTINGS parameters received from the server. Read-only.

Key Methods

Name	Description

Get	Sends a synchronous HTTP/3 GET request, returning the body as a string or writing it to a stream.
Post	Sends a synchronous HTTP/3 POST request with a request body.
Query	Sends a synchronous HTTP/3 QUERY request, a safe and idempotent verb that carries a body.
GetAsync	Sends a non-blocking GET; the reply arrives on OnResponse.
Extended-Connect	Performs an Extended CONNECT handshake (RFC 9220) and keeps the request stream open as a tunnel.
Discon-nect	Closes the HTTP/3 connection and the underlying QUIC transport.

Key Events

Name	Description
OnResponse	Fires when the client receives the full response (status, headers and body) from the server.
OnError	Fires when a request or connection error occurs. Essential when using the async methods.
OnHTTP3BeforeRequest	Fires before a request is sent, the hook for adding authorization or custom headers.
OnConnect	Fires after the HTTP/3 connection has been established.
OnDisconnect	Fires after the HTTP/3 connection has been closed.

Complete Example

The following example configures the client, sends a GET, a POST with a JSON body and a QUERY request, adds an authorization header to every request and closes the connection.

```
TsgcHTTP3Client *oClient = new TsgcHTTP3Client(NULL);
try
{
  oClient->ConnectTimeout = 10000;
  oClient->ReadTimeout = 30000;
  // refuse any response body larger than 10 MB
  oClient->MaxResponseSize = 10 * 1024 * 1024;
  oClient->TLSOptions->VerifyCertificate = true;
  oClient->OnHTTP3BeforeRequest = OnBeforeRequestEvent;

  UnicodeString vResponse = oClient->Get("https://www.example.com/");
  ShowMessage(vResponse);

  vResponse = oClient->Post("https://www.example.com/api/users",
    "{\"name\": \"John\"}", "application/json");
```

```
  ShowMessage(vResponse);

  // QUERY: a safe, idempotent request that carries a body
  vResponse = oClient->Query("https://www.example.com/api/search",
    "{\"city\": \"Madrid\"}", "application/json");
  ShowMessage(vResponse);

  oClient->Disconnect();
}
__finally
{
  delete oClient;
}
```

Requirements

- Requires the **All-Access** edition. HTTP/3 is delivered as the HTTP3 Pack, which is defined inside the All-Access block of `sgcVer.inc`.
- The pack ships **disabled**. Enable it by uncommenting `{$DEFINE SGC_PACK_HTTP3}` in `sgcVer.inc`, which in turn defines `SGC_QUIC`, `SGC_HTTP3` and `SGC_WEBTRANSPORT`.
- Those defines are nested inside `SGC_INDY_LIB`, so the Indy library must be enabled too. Without it the pack defines nothing and the component is not compiled.
- Add the unit `sgcHTTP` to the uses clause for `TsgcHTTP3Client`, and `sgcHTTP3_Classes` when you need the `TsgcHTTP3Response` and `TsgcHTTP3Settings` types.
- Requires **OpenSSL 3.5** or later, the first release providing native QUIC, when `TLSOptions.IOHandler = iohOpenSSL` (the default). The engine is then detected when OpenSSL is loaded: the built-in OpenSSL QUIC engine first, the sgc QUIC engine as the fallback. With `TLSOptions.IOHandler = iohNativeTLS` the sgc engine, the native TLS 1.3 handshake and the native packet protection are used, and OpenSSL is not loaded. The sgc engine also opens the HTTP/3 control and QPACK streams and reuses one QUIC connection for later requests.
- HTTP/3 runs over **UDP**. Make sure outbound UDP on the target port, normally 443, is permitted by any firewall on the path.

TsgcHTTP3Server

TsgcHTTP3Server implements an HTTP/3 server (RFC 9114) that serves HTTP requests over QUIC.

Introduction

TsgcHTTP3Server implements an HTTP/3 server as defined in **RFC 9114**. HTTP/3 carries HTTP requests and responses over **QUIC**, a UDP-based transport with TLS 1.3 built into it, so every connection is encrypted and multiplexed without head-of-line blocking: a lost packet stalls only the request stream it belonged to, not the whole connection. By default the server runs on the native QUIC listener provided by OpenSSL 3.5 or later, and falls back to the sgc QUIC engine when that listener is not available (see QUICMode), so no external HTTP/3 library is required. The component owns the whole HTTP/3 stack above QUIC: it opens the control and QPACK streams, exchanges SETTINGS, decodes QPACK header sections, reassembles the request frames on each stream and raises one event per complete request. Requests are multiplexed, so several may be in flight on a single connection, each on its own QUIC stream.
TLS is mandatory. QUIC has no plaintext mode, so the server cannot start without a certificate and a matching private key. Clients validate that certificate exactly as they would an HTTPS certificate, which means a self-signed certificate works only with clients configured to trust it. HTTP/3 also runs over UDP rather than TCP, so the listening port must be open for UDP in any firewall in between.
Follow the steps below to configure this component:
1. Create a new instance of **TsgcHTTP3Server** and set the Port the server will listen on. HTTP/3 uses UDP, and the inherited default port is not meaningful for HTTP/3, so always set it. Port `443` is the usual choice in production, since it lets the same authority be advertised through the `Alt-Svc` header of an HTTPS site.
2. Set the certificate and private key in QUICOptions.SSLOptions. Both are PEM files and both are required, because QUIC always negotiates TLS 1.3.
3. Handle the OnH3Request event and call Start. The aResponse parameter is created by the server with StatusCode preset to 200; set its StatusCode, ContentType and Data, and the server QPACK-encodes and sends it automatically when the handler returns. There is no need to call SendResponse. example:

```
TsgcHTTP3Server *oServer = new TsgcHTTP3Server(NULL);
oServer->Port = 443;
oServer->QUICOptions->SSLOptions->CertFile = "cert.pem";
oServer->QUICOptions->SSLOptions->KeyFile = "key.pem";
oServer->OnH3Request = OnH3RequestEvent;
oServer->Start();

void __fastcall OnH3RequestEvent(TObject *Sender, const
    TsgcQUICServerConnection *aConnection, const TsgcHTTP3Request *aRequest,
    const TsgcHTTP3Response *aResponse)
{
  aResponse->StatusCode = 200;
  aResponse->ContentType = "text/plain; charset=utf-8";
  aResponse->Data = TEncoding::UTF8->GetBytes("Hello HTTP/3");
}
```

Building the response

A request is answered by filling the `TsgcHTTP3Response` the server hands to OnH3Request. The object is owned by the server, so fill it and do not free it.

Member	Type	Description
`StatusCode`	`Integer`	Sent as the `:status` pseudo-header. Preset to `200`.
`Content-Type`	`string`	Sent as the `content-type` header when not empty.
`Data`	`TBytes`	The response body. Encode strings explicitly, normally with `TEncoding.UTF8.GetBytes`.

`Headers`	`TStringList`	Extra response headers, one entry per header in `name=value` form.
`ContentLength`	`Int64`	Not used when sending: `content-length` is computed from the length of Data.

The server writes `:status` and adds `content-type` and `content-length` for you, so those never need to be added by hand. Custom headers use `name=value` and not `name: value`. Pseudo-headers and the connection-specific headers forbidden by RFC 9114 (`connection`, `keep-alive`, `proxy-connection`, `transfer-encoding` and `upgrade`) are dropped rather than sent. A response to a HEAD request is sent as a header section only, with the body suppressed automatically. See OnH3Request for the full request and response reference.

Server push, tunnels and graceful shutdown

Server push. Call PushPromise from inside OnH3Request to promise an associated resource and deliver it on a server-initiated push stream. It returns `False` when the client has granted no push budget through MAX_PUSH_ID, which is normal and must be checked. Current mainstream browsers do not implement HTTP/3 server push, so it is useful mainly with clients you control.
Extended CONNECT and tunnels. Answering a CONNECT request with a 2xx status turns its request stream into a tunnel: the stream stays open, subsequent DATA frames stop being a request body and arrive at OnH3TunnelData, and TunnelSend writes bytes back. Set `Settings.EnableConnectProtocol` to advertise Extended CONNECT (RFC 9220), which lets a client send a CONNECT carrying a `:protocol` pseudo-header. Setting `H3Settings.Settings.WebTransportMaxSessions` above zero additionally enables WebTransport and raises OnWebTransportSession for each session, which is accepted unless the handler rejects it.
Graceful shutdown. Stop closes connections immediately. To drain instead, send SendGoAway on each connection first: it tells the peer the last request stream the server will handle, so clients stop opening new streams and can retry the rest elsewhere, while in-flight requests finish. Call Stop once they have.

Reference

- Properties — 5 properties documented, 3 of them inherited from TsgcQUICServer.
- Methods — 9 methods documented, plus 4 inherited QUIC transport methods listed for reference.
- Events — 8 events documented, plus the 4 inherited QUIC transport events, which are wired internally and must not be assigned.

Key Properties

Name	Description
Port	UDP port the HTTP/3 server listens on.
QUICOptions	QUIC transport options, including the mandatory certificate and private key, and connection limits.
H3Settings	HTTP/3 protocol settings advertised to peers: QPACK and field-section limits, Extended CONNECT and WebTransport.
Active	Starts or stops the HTTP/3 server listener.

Key Methods

Name	Description
Start	Starts the QUIC listener and begins accepting HTTP/3 connections.

Stop	Stops the listener and closes all active connections.
SendGoAway	Sends a GOAWAY frame to start a graceful connection shutdown.
PushPromise	Pushes an associated resource to the client from inside OnH3Request.

Key Events

Name	Description
OnH3Request	Fires when a complete HTTP/3 request is received so the application can fill the response.
OnH3Connect	Fires when a client establishes an HTTP/3 connection.
OnH3Disconnect	Fires when an HTTP/3 connection is closed.
OnH3Exception	Fires when an exception is raised while servicing an HTTP/3 connection.

Complete Example

The following example starts an HTTP/3 server that routes a couple of paths and returns a JSON or a plain-text body, logs connections, and stops the server cleanly. The response is filled inside OnH3Request and sent automatically by the server.

```
// start
TsgcHTTP3Server *oServer = new TsgcHTTP3Server(NULL);
oServer->Port = 4433;
oServer->QUICOptions->SSLOptions->CertFile = "server.crt";
oServer->QUICOptions->SSLOptions->KeyFile = "server.key";
oServer->OnH3Request = OnH3RequestEvent;
oServer->OnH3Connect = OnH3ConnectEvent;
oServer->OnH3Disconnect = OnH3DisconnectEvent;
oServer->OnH3Exception = OnH3ExceptionEvent;
oServer->Start();

void __fastcall OnH3ConnectEvent(TObject *Sender, const
    TsgcQUICServerConnection *aConnection)
{
  Log("HTTP/3 client connected: " + aConnection->PeerIP + ":" +
    IntToStr(aConnection->PeerPort));
}

void __fastcall OnH3DisconnectEvent(TObject *Sender, const
    TsgcQUICServerConnection *aConnection, __int64 aCode,
    const UnicodeString aReason)
{
  Log("HTTP/3 client disconnected: " + aReason);
}

void __fastcall OnH3ExceptionEvent(TObject *Sender, const
    TsgcQUICServerConnection *aConnection, Exception *aException)
{
  Log("HTTP/3 error: " + aException->Message);
}

// answer a request
void __fastcall OnH3RequestEvent(TObject *Sender, const
    TsgcQUICServerConnection *aConnection, const TsgcHTTP3Request *aRequest,
    const TsgcHTTP3Response *aResponse)
{
  if (aRequest->Path == "/api/time")
  {
    aResponse->StatusCode = 200;
    aResponse->ContentType = "application/json";
    aResponse->Data = TEncoding::UTF8->GetBytes(
      "{\"time\":\"" + DateTimeToStr(Now()) + "\"}");
  }
  else
  {
    aResponse->StatusCode = 404;
```

```
    aResponse->ContentType = "text/plain; charset=utf-8";
    aResponse->Data = TEncoding::UTF8->GetBytes("Not Found");
  }
}

// stop
oServer->Stop();
delete oServer;
```

Requirements

Require-ment	Detail
Edition	**All-Access** edition.
Compiler define	HTTP/3 ships disabled. Uncomment `{$DEFINE SGC_PACK_HTTP3}` in `sgcVer.inc`, which enables `SGC_QUIC`, `SGC_HTTP3` and `SGC_WEBTRANSPORT`.
Indy	`SGC_INDY_LIB` must be enabled: the HTTP/3 defines are nested inside it.
Unit	`sgcHTTP3_Server`, or `sgcHTTP` for the registered `TsgcHTTP3Server` component.
OpenSSL	**OpenSSL 3.5** or later with `QUICOptions.SSLOptions.IOHandler = iohOpenSSL`. There is no fallback on an older OpenSSL. With `iohNativeTLS` OpenSSL is not needed.
Transport	UDP. The listening port must be open for UDP, not TCP.
.NET	The .NET server accepts QUIC connections (.NET 8.0 or later, MsQuic) and implements the HTTP/3 framing and QPACK layer, which it uses for the SETTINGS handshake and for Extended CONNECT / WebTransport. It still does not serve plain HTTP/3 requests: they are never routed to OnH3Request, and SendResponse, SendGoAway and SendPush are not wired to the framing layer, so they do nothing. Use Delphi/C++Builder to serve HTTP/3 requests.

WebTransport

WebTransport over HTTP/3 lets both peers open many independent streams inside a single QUIC connection, negotiated with one Extended CONNECT request.

Introduction

WebTransport is a transport protocol layered on top of **HTTP/3** (RFC 9114), specified by **draft-ietf-webtrans-http3**. sgcWebSockets implements draft-09. A client opens a session with a single **Extended CONNECT** request carrying the `:protocol = webtransport` pseudo-header. Once the server answers `:status = 200`, both sides may open arbitrarily many **bidirectional** and **unidirectional** streams inside that session with no further round-trips. Because HTTP/3 runs over **QUIC**, those streams are independent and free from head-of-line blocking.
Think of it as the low-level sibling of WebSockets. Where a WebSocket gives you one ordered message channel, a WebTransport session gives you a whole connection worth of channels that you open and close as you please.

Extended CONNECT

WebTransport rides on **Extended CONNECT**, defined for HTTP/2 by RFC 8441 and ported to HTTP/3 by **RFC 9220**. A normal `CONNECT` request has no `:protocol` pseudo-header and no `:path`. An Extended CONNECT adds both, so a single request can bootstrap an arbitrary protocol on top of one request stream. For WebTransport the request looks like this:

```
:method    = CONNECT
:protocol  = webtransport
:scheme    = https
:authority = example.com
:path      = /echo
```

The request stream stays open for the lifetime of the session and is never used for HTTP body data. Its QUIC stream id becomes the **session id**.

The SETTINGS that enable it

A peer must advertise two HTTP/3 SETTINGS before WebTransport may be used:

Setting	Meaning
`SETTINGS_ENABLE_CONNECT_PROTOCOL`	RFC 9220. The peer accepts the `:protocol` pseudo-header, so Extended CONNECT is legal on this connection.
`SETTINGS_WEBTRANSPORT_MAX_SESSIONS`	draft-ietf-webtrans-http3 Section 3.1. The number of concurrent WebTransport sessions the peer will accept. A value of `0` means WebTransport is not offered.

On the TsgcHTTP3Server, setting `H3Settings.Settings.WebTransportMaxSessions` to a value greater than zero advertises both settings automatically and switches the connection into multi-stream mode. On the TsgcHTTP3Client, `ServerSupportsWebTransport` returns True once the peer SETTINGS frame has been read and it carried both values.

The object model

Two classes model a session, and both live in the `sgcWebTransport` unit:

- TsgcWebTransportSession — one session. It owns its streams, carries the session id, and raises the session events. You never construct one yourself: the client returns it from `WebTransportConnect`, the server hands it to `OnWebTransportSession`.
- TsgcWebTransportStream — one stream inside a session. Open one with `Session.OpenStream`, or receive a peer-opened one through `Session.OnStream`.

Every WebTransport stream is a real QUIC stream prefixed with a small header that binds it to a session. A unidirectional stream starts with `varint(0x54) + varint(SessionId)`, a bidirectional one with `varint(0x41) + varint(SessionId)`. The library writes and parses that header for you: `WriteData` carries application bytes only, and `OnData` delivers application bytes only.

Server Flow

Enable WebTransport by setting `H3Settings.Settings.WebTransportMaxSessions` above zero, then handle `OnWebTransportSession`. The handler receives a live session and an accept flag, `var aAccept` in Delphi and C++Builder, `e.Accept` on .NET. It arrives as **True** on every stack, so doing nothing accepts the session; set it to `False` to reject, and the server replies `:status = 404` and discards the session. When accepted, the server replies `:status = 200`, keeps the CONNECT stream open and registers the session.
Assign `OnStream` inside the handler, before returning, because peer-opened streams can arrive as soon as the session is registered.

```
// WebTransport is enabled only when WebTransportMaxSessions > 0
FServer = new TsgcHTTP3Server(NULL);
FServer->H3Settings->Settings->WebTransportMaxSessions = 4;
FServer->OnWebTransportSession = OnWTSession;

void __fastcall TfrmServer::OnWTSession(TObject *Sender,
  TsgcQUICServerConnection * const aConnection,
  TsgcWebTransportSession * const aSession, bool &aAccept)
{
  aAccept = true;   // already true on entry; set false to reject with 404
  aSession->OnStream = OnWTStream;
  aSession->OnClose = OnWTSessionClose;
}

void __fastcall TfrmServer::OnWTStream(TObject *Sender,
  TsgcWebTransportStream * const aStream)
{
  aStream->OnData = OnWTStreamData;
}

void __fastcall TfrmServer::OnWTStreamData(TObject *Sender, const TBytes aData)
{
  // echo the bytes back on the same stream
  static_cast<TsgcWebTransportStream *>(Sender)->WriteData(aData);
}
```

Client Flow

Call `WebTransportConnect` with the session URL. It opens a dedicated QUIC connection, sends the Extended CONNECT, waits for the response bounded by `ConnectTimeout`, and returns the session on `:status = 200` or `nil` otherwise. It does not require a prior `Connect` call: if the client is already connected, it disconnects first, because a WebTransport session needs the CONNECT request to travel on the first client-initiated bidirectional stream.
The WebTransport connection is non-blocking, so nothing is read from the socket unless you pump it. Call `WebTransportProcessEvents` periodically from a timer or a background thread. Without it, `OnData` and `OnStream` never fire.

```
FClient = new TsgcHTTP3Client(NULL);
FClient->TLSOptions->VerifyCertificate = false;
FClient->ConnectTimeout = 10000;

// Opens the QUIC connection and performs the Extended CONNECT. Returns NULL if
// the server rejects the session or does not support WebTransport.
FSession = FClient->WebTransportConnect(L"https://127.0.0.1:4433/echo");
if (FSession != NULL)
{
  FStream = FSession->OpenStream(false);   // false = bidirectional
  FStream->OnData = OnStreamData;
  FStream->WriteData(L"Hello WebTransport");
  tmrPump->Enabled = true;
}

void __fastcall TfrmClient::tmrPumpTimer(TObject *Sender)
{
  // Pump the QUIC connection so incoming stream data is harvested
  FClient->WebTransportProcessEvents();
}
```

Datagrams: current status

WebTransport as specified also offers unreliable datagrams, and this library exposes the API for them: SendDatagram on the session and the OnDatagram event.
As shipped, WebTransport in sgcWebSockets is streams only. Datagrams do not reach the wire and do not arrive from it. This is true on the client, on the server, in Delphi, in C++Builder and on .NET. Streams work fully; datagrams do not work at all. The outbound call fails loudly rather than quietly, which is the one thing that changed. Concretely:

- **Outbound.** In Delphi and C++Builder `SendDatagram` raises `TsgcQUICException`. It checks DatagramsSupported, the flag by which a transport declares that its datagram writer hook (OnSendDatagramData) genuinely reaches the wire. No shipping transport sets it: the HTTP/3 client binds that hook to a deliberately empty handler, and the HTTP/3 server binds it to `TsgcHTTP3Server.SendDatagram`, whose base implementation is a deliberate no-op. So the method refuses before encoding anything, rather than lose your payload in silence. On .NET the managed `SendDatagram` does not raise, it discards its argument directly.
- **Inbound.** `OnDatagram` is raised only by `TsgcWebTransportSession.DoDatagram`. Its only caller is `TsgcHTTP3ServerConnection.ProcessWebTransportDatagram`, an integration point for a datagram-capable transport that the shipping HTTP/3 stack never calls. `OnDatagram` therefore never fires.

Why, and on which engine. sgcWebSockets has two QUIC engines, `TsgcQUICMode = (qmCustom, qmBuiltin)`, chosen automatically when OpenSSL loads and not selectable by the application. `qmCustom` is the Pascal QUIC stack, used when the loaded library exposes the quictls QUIC TLS API. `qmBuiltin` is **OpenSSL's own native QUIC**, used otherwise.
WebTransport runs on `qmBuiltin` in every shipping configuration. The bundled OpenSSL 3.5 exports `OSSL_QUIC_client_method` but not the quictls QUIC TLS entry points, so `qmBuiltin` is what gets selected. It is also the only viable choice: `qmCustom` has no server support and cannot open the extra QUIC streams a WebTransport session needs.
And `qmBuiltin` is precisely the engine with no datagram path. WebTransport datagrams need the QUIC DATAGRAM extension (RFC 9221) underneath the HTTP Datagram layer (RFC 9297). OpenSSL's native QUIC does not implement RFC 9221, and its server side exposes no DATAGRAM write path, so there is nowhere to put the bytes. Writing them onto a reliable stream instead would corrupt the HTTP/3 framing, so the library refuses the call rather than break the connection. On .NET the cause is the same in a different guise: `System.Net.Quic` does not expose raw QUIC datagrams.
Yet the server still advertises SETTINGS_H3_DATAGRAM. Enabling WebTransport makes the server advertise it, together with SETTINGS_ENABLE_CONNECT_PROTOCOL and SETTINGS_WEBTRANSPORT_MAX_SESSIONS, even though no datagram can be carried. That is deliberate, not a stale setting. The WebTransport over HTTP/3 draft requires the setting, and browsers refuse to establish a session without it, so it is advertised for interoperability: without it the stream half of WebTransport, the half that works, would be unreachable from a browser.
What is there. The full encode and demultiplex path in `sgcWebTransport` is implemented and correct: Quarter-Stream-ID encode and decode, the session lookup by Quarter-Stream-ID, and the payload delivery. A datagram-capable QUIC backend can light datagrams up by supplying a real writer (assigning `OnSendDatagramData`, or overriding the virtual `TsgcHTTP3Server.SendDatagram`), setting `DatagramsSupported := True` on each session so the encoder runs instead of raising, and calling `ProcessWebTransportDatagram` on receipt. Until such a backend exists, treat datagram support as absent, and design your protocol on streams.

Requirements

- In `sgcVer.inc`, the HTTP/3 feature pack `SGC_PACK_HTTP3` is declared inside the **All-Access** edition block (`SGC_EDT_ALL`) and requires `SGC_INDY_LIB`.
- HTTP/3 and WebTransport ship **disabled**. Uncomment `{$DEFINE SGC_PACK_HTTP3}` in `sgcVer.inc`. That in turn defines `SGC_QUIC`, `SGC_HTTP3` and `SGC_WEBTRANSPORT`. All WebTransport code is guarded by `{$IFDEF SGC_WEBTRANSPORT}`.
- Units: `sgcWebTransport` for the session and stream classes, `sgcHTTP` for `TsgcHTTP3Client` and `TsgcHTTP3Server`, `sgcQUIC_Server` for `TsgcQUICServerConnection`.
- Requires **OpenSSL 3.5** or later. The QUIC engine (`qmCustom` or `qmBuiltin`) is auto-detected when OpenSSL loads and is not user-selectable. WebTransport runs on `qmBuiltin`, OpenSSL's native QUIC, which is what the bundled OpenSSL 3.5 provides and the only engine with the server and multi-stream support a session needs. That engine has no RFC 9221 DATAGRAM path, which is why datagrams are inert and `SendDatagram` raises.
- .NET: available in the pure-managed library on **net8.0** or later with MsQuic present. `WebTransportConnect` raises `PlatformNotSupportedException` on netstandard2.0 or when `QuicConnection.IsSupported` is False.

Reference

- TsgcWebTransportSession — one WebTransport session; returned by the client's WebTransportConnect, handed to the server's OnWebTransportSession.
- TsgcWebTransportStream — one bidirectional or unidirectional stream inside a session.
- TsgcHTTP3Client — the HTTP/3 client that opens WebTransport sessions.
- TsgcHTTP3Server — the HTTP/3 server that accepts them.
- HTTP/3 — the protocol WebTransport is layered on.

TsgcWebTransportSession

TsgcWebTransportSession represents one WebTransport session and owns the streams multiplexed inside it over a single HTTP/3 QUIC connection.

Introduction

A WebTransport session is established by one **Extended CONNECT** request (RFC 9220) carrying `:protocol = webtransport`. TsgcWebTransportSession is the object that represents it once the handshake succeeds. It is declared in the `sgcWebTransport` unit and it is a plain class, not a component: you do not drop it on a form and you do not construct it. It reaches you in one of two ways:

- **Client.** `TsgcHTTP3Client.WebTransportConnect` returns the session on `:status = 200`, or `nil` when the server rejects it.
- **Server.** `TsgcHTTP3Server.OnWebTransportSession` hands you the session together with a `var aAccept` flag.

In both cases the session is owned by the connection's session registry, which frees it. Do not free it yourself. On the client, freeing the `TsgcHTTP3Client` releases every session it opened, and each session releases its streams.
The session multiplexes any number of TsgcWebTransportStream objects. Open one with OpenStream; streams the peer opens arrive on OnStream. The session itself carries no read or write method for application data: all payload travels on streams.

Session identity

SessionId is the QUIC stream id of the Extended CONNECT request that created the session. It anchors the whole session on the wire: every WebTransport stream opened inside it is prefixed with `varint(0x54)` or `varint(0x41)` followed by `varint(SessionId)`, which is how the receiving side knows which session a new QUIC stream belongs to. QuarterStreamId is the same identity in the compact form the datagram layer uses, `SessionId div 4`.

Streams only

The session exposes SendDatagram and OnDatagram, but no shipping backend can carry a datagram: `SendDatagram` raises `TsgcQUICException` rather than lose your payload silently, and `OnDatagram` never fires. See WebTransport for the full explanation. Build your protocol on streams.

Reference

- Properties — 4 properties, 3 of them read-only.
- Methods — 8 public methods.
- Events — 6 events, 3 for applications and 3 transport hooks.

Key Properties

Name	Description
SessionId	QUIC stream id of the Extended CONNECT request that established this session.
QuarterStreamId	SessionId div 4, the session identity as encoded in WebTransport datagrams.
StreamCount	Number of streams currently open on this session.

DatagramsSupported	Whether the transport bound to this session can really put a datagram on the wire. False on every shipping transport.

Key Methods

Name	Description
OpenStream	Opens a new outgoing stream, bidirectional or unidirectional, and sends its WebTransport header.
FindStreamById	Returns the stream with the given QUIC stream id, or nil.
Close	Closes the session's open streams and raises OnClose. It does not tear down the QUIC connection.
SendDatagram	Sends a WebTransport datagram. Raises TsgcQUICException as shipped, since no transport can carry one.

Key Events

Name	Description
OnStream	Fires when the peer opens a new stream inside this session.
OnClose	Fires when Close is called on this session.
OnDatagram	Would deliver an inbound datagram. Never fires as shipped.

Complete Example

A full client round-trip against a WebTransport echo server: connect, open a bidirectional stream, write, read the echo through `OnData`, then close. The QUIC connection is non-blocking, so a timer pumps it with `WebTransportProcessEvents`; without that pump no data is ever read.

```
// --- FWebTransportClient.h ---
#include <sgcHTTP.hpp>
#include <sgcHTTP3_Classes.hpp>
#include <sgcHTTP3_Types.hpp>
#include <sgcWebTransport.hpp>

class TfrmClient : public TForm
{
__published:
  TMemo *memoLog;
  TTimer *tmrPump;
  TButton *btnConnect;
  TButton *btnClose;
  void __fastcall FormCreate(TObject *Sender);
  void __fastcall FormDestroy(TObject *Sender);
  void __fastcall btnConnectClick(TObject *Sender);
  void __fastcall btnCloseClick(TObject *Sender);
  void __fastcall tmrPumpTimer(TObject *Sender);
private:
  TsgcHTTP3Client *FClient;
  TsgcWebTransportSession *FSession;
  TsgcWebTransportStream *FStream;
  void __fastcall OnStreamData(TObject *Sender, const TBytes aData);
  void __fastcall OnStreamClose(TObject *Sender);
  void __fastcall OnSessionClose(TObject *Sender);
};
```

```
// --- FWebTransportClient.cpp ---
void __fastcall TfrmClient::FormCreate(TObject *Sender)
{
  FClient = new TsgcHTTP3Client(NULL);
  FClient->TLSOptions->VerifyCertificate = false;   // self-signed test cert
  FClient->ConnectTimeout = 10000;
  FClient->UserAgent = L"sgcWebSockets/WebTransportClient";

  tmrPump->Interval = 50;
  tmrPump->Enabled = false;
}

void __fastcall TfrmClient::FormDestroy(TObject *Sender)
{
  tmrPump->Enabled = false;
  // FSession is owned by FClient->WebTransportSessions and FStream is owned by
  // the session, so deleting the client releases both.
  delete FClient;
}

void __fastcall TfrmClient::btnConnectClick(TObject *Sender)
{
  // Opens a dedicated QUIC connection and performs the Extended CONNECT.
  // Returns NULL when the server rejects the session.
  FSession = FClient->WebTransportConnect(L"https://127.0.0.1:4433/echo");
  if (FSession == NULL)
  {
    memoLog->Lines->Add(L"WebTransport connect failed.");
    return;
  }

  memoLog->Lines->Add(L"session established: id=" +
    IntToStr(FSession->SessionId));
  FSession->OnClose = OnSessionClose;

  // false = bidirectional, so the server can answer on the same stream.
  FStream = FSession->OpenStream(false);
  FStream->OnData = OnStreamData;
  FStream->OnClose = OnStreamClose;
  memoLog->Lines->Add(L"bidi stream opened: id=" +
    IntToStr(FStream->StreamId));

  // Start pumping BEFORE writing, so the echo cannot be missed.
  tmrPump->Enabled = true;

  FStream->WriteData(L"Hello WebTransport");   // UTF-8 encoded for you
}

void __fastcall TfrmClient::tmrPumpTimer(TObject *Sender)
{
  // Reads peer packets and harvests peer-opened streams.
  FClient->WebTransportProcessEvents();
}

void __fastcall TfrmClient::OnStreamData(TObject *Sender, const TBytes aData)
{
  TsgcWebTransportStream *oStream =
    static_cast<TsgcWebTransportStream *>(Sender);
  memoLog->Lines->Add(L"echo on stream " + IntToStr(oStream->StreamId) +
    L": " + TEncoding::UTF8->GetString(aData));
}

void __fastcall TfrmClient::OnStreamClose(TObject *Sender)
{
  memoLog->Lines->Add(L"stream closed");
}

void __fastcall TfrmClient::OnSessionClose(TObject *Sender)
{
  memoLog->Lines->Add(L"session closed");
}

void __fastcall TfrmClient::btnCloseClick(TObject *Sender)
{
  tmrPump->Enabled = false;
  if (FStream != NULL)
    FStream->Close();     // fires OnStreamClose
  if (FSession != NULL)
    FSession->Close();    // fires OnSessionClose
}
```

Requirements

- In `sgcVer.inc`, the HTTP/3 feature pack `SGC_PACK_HTTP3` is declared inside the **All-Access** edition block (`SGC_EDT_ALL`) and requires `SGC_INDY_LIB`.
- HTTP/3 and WebTransport ship **disabled**. Uncomment `{$DEFINE SGC_PACK_HTTP3}` in `sgcVer.inc`, which defines `SGC_QUIC`, `SGC_HTTP3` and `SGC_WEBTRANSPORT`. The whole class is guarded by `{$IFDEF SGC_WEBTRANSPORT}`.
- Units: `sgcWebTransport` declares the class; add `sgcHTTP` for `TsgcHTTP3Client` and `TsgcHTTP3Server`, and `sgcQUIC_Server` for `TsgcQUICServerConnection`.
- Requires **OpenSSL 3.5** or later. WebTransport runs on the `qmBuiltin` QUIC engine (OpenSSL's native QUIC), which is auto-detected at OpenSSL load and is not user-selectable.
- .NET: **net8.0** or later with MsQuic. Namespace `esegece.sgcWebSockets`.
- Streams only. `qmBuiltin` has no RFC 9221 DATAGRAM path, so datagrams cannot be sent in any shipping configuration and `SendDatagram` raises.

TsgcWebTransportStream

TsgcWebTransportStream represents one bidirectional or unidirectional stream inside a WebTransport session. It is where all application data travels.

Introduction

A WebTransport session carries no payload of its own. Everything you send and receive travels on a stream, and TsgcWebTransportStream is that stream. It wraps one real QUIC stream, so streams within a session are independent: a stall or a loss on one does not hold up the others, which is the head-of-line-blocking freedom QUIC exists to provide. The class is declared in the `sgcWebTransport` unit and, like TsgcWebTransportSession, it is a plain class rather than a component. You never construct one. It reaches you in one of two ways:

- **You opened it.** `Session.OpenStream` returns a stream ready to write.
- **The peer opened it.** The stream arrives as the `aStream` parameter of the session's `OnStream` event.

Either way, the Session owns the stream and frees it. Do not free it yourself.

Two directions

Unidirectional is fixed when the stream is opened and says which kind you have:

Kind	Header on the wire	Behaviour
Bidirectional (`Unidirectional = False`)	`varint(0x41) + varint(SessionId)`	Both sides can write. The natural choice for request and response, or for an echo.
Unidirectional (`Unidirectional = True`)	`varint(0x54) + varint(SessionId)`	Only the opener can write. The receiver's copy is receive-only, so writing to it does nothing useful.

That header is written once, by `OpenStream`, before you get the stream. Everything you pass to WriteData is pure application payload, and everything OnData gives you is pure application payload. You never see the framing.

It is a byte stream, not a message queue

This is the single most common source of bugs. `WriteData` and `OnData` do not preserve message boundaries. QUIC delivers a stream of bytes, and one `WriteData` may surface as several `OnData` calls, or several writes may arrive coalesced into one. If your protocol has messages, frame them yourself, with a length prefix, a delimiter, or fixed-size records. The `string` overload of `WriteData` encodes to UTF-8, and short test payloads usually arrive in one piece, which makes this trap easy to miss until traffic gets real.

Closing is a local signal

Close marks the stream closed, unregisters it from its session, and raises OnClose, so `StreamCount` drops. It does not send a QUIC FIN and does not tell the peer. Like the session's `Close`, it is guarded: calling it twice raises `OnClose` once.

Reference

- Properties — 4 properties.
- Methods — 4 public methods.
- Events — 3 events, 2 for applications and 1 transport hook.

Key Properties

Name	Description
StreamId	The QUIC stream id backing this stream.
Unidirectional	True for a one-way stream, False for a bidirectional one. Fixed at construction.
Session	The session that owns this stream.

Key Methods

Name	Description
WriteData	Writes application bytes to the stream. Overloaded for TBytes and for string (UTF-8 encoded).
Close	Unregisters the stream from its session and raises OnClose. A local signal only.

Key Events

Name	Description
OnData	Fires when application bytes arrive on this stream.
OnClose	Fires when Close is called on this stream.

Complete Example

A WebTransport echo server: accept the session, take the streams the client opens, and echo every payload back on the same stream. This is the server side of the round-trip shown on the TsgcWebTransportSession page, so the two can be run against each other.

```
// --- FWebTransportServer.h ---
#include <sgcHTTP.hpp>
#include <sgcHTTP3_Classes.hpp>
#include <sgcHTTP3_Types.hpp>
#include <sgcQUIC_Server.hpp>
#include <sgcWebTransport.hpp>

class TfrmServer : public TForm
{
__published:
  TMemo *memoLog;
  TButton *btnStart;
  TButton *btnStop;
  void __fastcall FormCreate(TObject *Sender);
  void __fastcall FormDestroy(TObject *Sender);
  void __fastcall btnStartClick(TObject *Sender);
  void __fastcall btnStopClick(TObject *Sender);
private:
  TsgcHTTP3Server *FServer;
  void __fastcall OnWTSession(TObject *Sender,
    TsgcQUICServerConnection * const aConnection,
    TsgcWebTransportSession * const aSession, bool &aAccept);
  void __fastcall OnWTStream(TObject *Sender,
    TsgcWebTransportStream * const aStream);
  void __fastcall OnWTStreamData(TObject *Sender, const TBytes aData);
  void __fastcall OnWTStreamClose(TObject *Sender);
};

// --- FWebTransportServer.cpp ---
void __fastcall TfrmServer::FormCreate(TObject *Sender)
{
```

```
  FServer = new TsgcHTTP3Server(NULL);

  // WebTransport is enabled ONLY when this is greater than zero. Setting it
  // advertises SETTINGS_ENABLE_CONNECT_PROTOCOL and
  // SETTINGS_WEBTRANSPORT_MAX_SESSIONS, and switches the connection into
  // multi-stream mode.
  FServer->H3Settings->Settings->WebTransportMaxSessions = 4;

  FServer->OnWebTransportSession = OnWTSession;
}

void __fastcall TfrmServer::FormDestroy(TObject *Sender)
{
  delete FServer;   // frees every session and every stream underneath
}

void __fastcall TfrmServer::btnStartClick(TObject *Sender)
{
  FServer->Host = L"0.0.0.0";
  FServer->Port = 4433;
  FServer->QUICOptions->SSLOptions->CertFile = L"server.crt";
  FServer->QUICOptions->SSLOptions->KeyFile = L"server.key";
  FServer->Start();
}

void __fastcall TfrmServer::btnStopClick(TObject *Sender)
{
  FServer->Stop();
}

void __fastcall TfrmServer::OnWTSession(TObject *Sender,
  TsgcQUICServerConnection * const aConnection,
  TsgcWebTransportSession * const aSession, bool &aAccept)
{
  // aAccept arrives true, so doing nothing accepts. Set false to reject,
  // and the server answers :status=404 and discards the session.
  aAccept = true;

  memoLog->Lines->Add(L"session " + IntToStr(aSession->SessionId) +
    L" accepted from " + aConnection->PeerIP + L":" +
    IntToStr(aConnection->PeerPort));

  // Assign OnStream BEFORE returning: the client may open a stream the
  // moment it sees our :status=200.
  aSession->OnStream = OnWTStream;
}

void __fastcall TfrmServer::OnWTStream(TObject *Sender,
  TsgcWebTransportStream * const aStream)
{
  // Assign OnData FIRST. Bytes that arrived with the stream header are
  // delivered as soon as this handler returns, and are lost otherwise.
  aStream->OnData = OnWTStreamData;
  aStream->OnClose = OnWTStreamClose;

  memoLog->Lines->Add(L"stream " + IntToStr(aStream->StreamId) +
    L" opened by peer (unidirectional=" +
    BoolToStr(aStream->Unidirectional, true) + L")");
}

void __fastcall TfrmServer::OnWTStreamData(TObject *Sender, const TBytes aData)
{
  TsgcWebTransportStream *oStream =
    dynamic_cast<TsgcWebTransportStream *>(Sender);
  if (oStream == NULL)
    return;

  memoLog->Lines->Add(L"stream " + IntToStr(oStream->StreamId) +
    L" received " + IntToStr(aData.Length) + L" bytes: " +
    TEncoding::UTF8->GetString(aData));

  // ECHO. Only a bidirectional stream can be written back on: a peer-opened
  // unidirectional stream is receive-only at this end.
  if (!oStream->Unidirectional)
  {
    oStream->WriteData(aData);
    memoLog->Lines->Add(L"echoed " + IntToStr(aData.Length) + L" bytes");
  }
}

void __fastcall TfrmServer::OnWTStreamClose(TObject *Sender)
{
  memoLog->Lines->Add(L"stream closed");
}
```

Requirements

- In `sgcVer.inc`, the HTTP/3 feature pack `SGC_PACK_HTTP3` is declared inside the **All-Access** edition block (`SGC_EDT_ALL`) and requires `SGC_INDY_LIB`.
- HTTP/3 and WebTransport ship **disabled**. Uncomment `{$DEFINE SGC_PACK_HTTP3}` in `sgcVer.inc`, which defines `SGC_QUIC`, `SGC_HTTP3` and `SGC_WEBTRANSPORT`. The whole class is guarded by `{$IFDEF SGC_WEBTRANSPORT}`.
- Units: `sgcWebTransport` declares the class; add `sgcHTTP` for `TsgcHTTP3Client` and `TsgcHTTP3Server`, and `sgcQUIC_Server` for `TsgcQUICServerConnection`.
- Requires **OpenSSL 3.5** or later. WebTransport runs on the `qmBuiltin` QUIC engine (OpenSSL's native QUIC), auto-detected at OpenSSL load and not user-selectable.
- .NET: **net8.0** or later with MsQuic. Namespace `esegece.sgcWebSockets`.
- Streams are fully supported. Datagrams are not: see WebTransport.

gRPC

gRPC is a high-performance, contract-based Remote Procedure Call (RPC) framework. It runs over HTTP/2 and uses Protocol Buffers as its interface definition language and message serialization format, so clients and servers share a strongly-typed contract and exchange compact binary messages.

Main features

- Runs over HTTP/2, so multiple calls share a single connection through multiplexed streams.
- Supports the four gRPC RPC patterns: unary, server-streaming, client-streaming and bidirectional streaming.
- Uses Protocol Buffers to encode request and response messages as compact binary data.
- Supports metadata (custom headers), deadlines and message compression.
- Provides pluggable interceptors, automatic retries and client-side load balancing.
- Implements the standard gRPC Health Checking and Server Reflection services.

Components

- **TsgcGRPCClient:** gRPC client component that runs over TsgcHTTP2Client and supports the four gRPC RPC patterns (unary, server-streaming, client-streaming and bidirectional streaming).

Tools

sgcProtoBuf is a code generator that reads a Protocol Buffers definition and produces Delphi units with a class for every message. It resolves the imported definitions, generates one unit per package and maps the google.protobuf types onto the types shipped with the library. See sgcProtoBuf for details.

Google Cloud Interfaces

The library ships with typed interfaces built on top of TsgcGRPCClient for calling Google Cloud gRPC APIs, including Pub/Sub, Speech-to-Text, Translation, Vision, Natural Language, Cloud Storage, BigQuery Storage and Vertex AI. See Google Cloud gRPC services for details.

sgcProtoBuf

sgcProtoBuf is a code generator shipped with the library. It reads a Protocol Buffers definition, a **.proto** file, and produces Delphi units with a class for every message, so those messages can be serialized and deserialized in Delphi. It supports proto2 and proto3, and it reads the syntax from the file itself. The generator is available as a wizard application and as a command line tool.

Wizard

The wizard application generates the units in a few steps:

1. Select the Protocol Buffers definition, either a local **.proto** file or a URL.
2. The tool validates the syntax and reports whether it is supported.
3. Set the generation options.
4. Review the messages, enums, services and imports that were found in the definition.
5. Finish and choose where the generated units are written.

Options

The following options control what is generated:

- **Classname:** the prefix used for the generated types.
- **Namespace:** the namespace applied to the generated units.
- **Generate classes:** generates a class for every message declared in the definition.
- **Generate service stubs:** generates the stubs for the services declared in the definition.
- **Generate documentation comments:** copies the comments of the definition into the generated code.
- **Resolve imports:** generates one unit per package for the whole dependency tree.
- **Import root:** the folder searched for imported **.proto** files. When left empty, it is discovered automatically from the selected file.
- **Generate enums as Delphi enumerated types:** generates every enum as a Delphi enumerated type instead of a constant block.
- **Keep original proto element names:** uses the names declared in the definition, without the **Tsgc** prefix.
- **Short enum constant names:** generates the enum constants without the enum name prefix.

Unit Names

The generated unit name comes from the **package** statement of the **.proto** file, so a definition that declares **package avl.ditest.counter.v1;** produces **unit avl.ditest.counter.v1;** and is saved as **avl.ditest.counter.v1.pas**. The file name always matches the unit name. Several **.proto** files that declare the same package are generated into a single unit.

Imports

When **resolve imports** is enabled, the tool follows the **import** statements recursively, generates every dependency and adds the generated units to the uses clause. An import that cannot be found under the import root is reported in the message list, instead of producing a unit that does not compile.

Well-Known Types

The **google.protobuf** types are mapped onto the classes already shipped in the **sgcProtoBuf_WellKnownTypes** unit, so the Google **.proto** files never need to be generated. The mapped types include:

- **Timestamp** and **Duration**.
- **Any**, **Empty** and **FieldMask**.
- **Struct**, together with Value and ListValue.
- The wrapper types, such as **StringValue** and **Int32Value**.

Enums

With the enumerated types option enabled, an enum declared in the definition becomes a real Delphi enumerated type, so **enum CountryCodeDto** is generated as:

```
// Enum: CountryCodeDto
TsgcGRPCHelloworldCountryCodeDto = (
  COUNTRY_CODE_DTO_UNSPECIFIED = 0,
  COUNTRY_CODE_DTO_ES = 1,
  COUNTRY_CODE_DTO_FR = 2
);
```

A Delphi enumerated type has limitations that a Protocol Buffers enum does not have. A Delphi enumerated type cannot hold negative ordinals or duplicate values, and two enums declared in the same unit cannot declare the same value name. In those cases the generator automatically falls back to a constant block for that enum and adds a comment explaining why. The rest of the enums in the same file are not affected.

Generated Code

Every message class descends from a generated base class and exposes **LoadFromBytes**, **LoadFromStream** and **ToBytes**, plus **DoRead** and **DoWrite**. A property is generated for every field declared in the message. Example:

```
oRequest := TsgcGRPCHelloworldHelloRequest.Create;
Try
  oRequest.LoadFromBytes(vReceived);
  ShowMessage(oRequest.Name);
  vBytes := oRequest.ToBytes;
Finally
  oRequest.Free;
End;
```

Command Line

The command line tool accepts the following switches:

- **--input=<file>:** the **.proto** file to read.
- **--output=<file>:** the unit file to write.
- **--output-dir=<dir>:** the folder where the generated units are written.
- **--classname=<name>:** the prefix used for the generated types.
- **--namespace=<name>:** the namespace applied to the generated units.
- **--no-classes:** does not generate the message classes.
- **--no-services:** does not generate the service stubs.
- **--no-docs:** does not generate the documentation comments.
- **--enum-as-type:** generates the enums as Delphi enumerated types.
- **--original-names:** keeps the original proto element names.
- **--short-enum-consts:** generates the enum constants without the enum name prefix.
- **--resolve-imports:** follows the import statements and generates the whole dependency tree.
- **--import-root=<dir>:** the folder searched for imported **.proto** files. It may be repeated to search more than one root folder.
- **--log:** writes the generation log.
- **--help:** shows the list of switches.

Six of the switches also accept a short form: **-i** for **--input**, **-o** for **--output**, **-c** for **--classname**, **-n** for **--namespace**, **-d** for **--output-dir** and **-h** for **--help**.
Example, generating a whole dependency tree into the **units** folder, with the enums generated as Delphi enumerated types:

```
sgcProtoBuf.exe --input=protos\helloworld.proto --resolve-imports --output-dir=units --enum-as-type
```

See Also

- gRPC

TsgcGRPCClient

TsgcGRPCClient implements a gRPC client over HTTP/2 (using TsgcHTTP2Client as transport) and supports the four gRPC RPC patterns.

Introduction

TsgcGRPCClient implements a gRPC client that runs over HTTP/2. Follow the steps below to configure this component:
1. Assign a **TsgcHTTP2Client** instance to the **Client** property; this component is used as the HTTP/2 transport. Set the Host, Port and TLS of the transport to point to the gRPC server.
2. Handle the responses using the **OnGRPCResponse** event, then connect and call a unary method. example:

```
TsgcHTTP2Client *oHTTP2 = new TsgcHTTP2Client();
oHTTP2->Host = "grpc.example.com";
oHTTP2->Port = 443;
oHTTP2->TLS = true;

TsgcGRPCClient *oGRPC = new TsgcGRPCClient();
oGRPC->Client = oHTTP2;
oGRPC->OnGRPCResponse = OnGRPCResponseEvent;
oGRPC->CallAsync("helloworld.Greeter", "SayHello", vRequestBytes);

void OnGRPCResponseEvent(TObject *Sender, const TsgcGRPCResponse *aResponse)
{
  if (aResponse->StatusCode == grpcOK)
    ShowMessage(aResponse->DataString);
}
```

RPC Patterns

gRPC defines four RPC patterns. TsgcGRPCClient exposes a set of members for each one:

- **Unary:** use **Call** for a synchronous call that returns a TsgcGRPCResponse, or **CallAsync** to receive the response asynchronously through the OnGRPCResponse event.
- **Server streaming:** use **ServerStreamingCall**; each message sent by the server arrives in OnGRPCStreamMessage and the stream completes in OnGRPCStreamEnd.
- **Client streaming:** use **OpenClientStream**, then **SendStreamMessage** for one or more messages, and finish with **CloseClientStream** or **CloseClientStreamAsync**.
- **Bidirectional streaming:** use **OpenBidiStream**, send messages with **SendBidiMessage** and finish with **CloseBidiStream**; incoming messages arrive in OnGRPCStreamMessage.

The following example performs a server-streaming call and reads every message delivered by the server:

```
oGRPC->OnGRPCStreamMessage = OnGRPCStreamMessageEvent;
oGRPC->ServerStreamingCall("routeguide.RouteGuide", "ListFeatures", vRequestBytes);

void OnGRPCStreamMessageEvent(TObject *Sender, const TsgcGRPCStreamMessage *aMessage)
{
  ShowMessage(aMessage->DataString);
}
```

Features

Channel Options

The ChannelOptions property configures channel-level behaviour such as message Compression, the Content-Type, the maximum send and receive message sizes and HTTP/2 keep-alive pings.

Metadata

Use DefaultMetadata to attach custom headers to every call, or pass per-call metadata to any of the call methods. This is how authorization headers such as a Bearer token are sent to the server.

Deadlines and Retry

Each call accepts a deadline so it fails fast when the server is slow. The RetryPolicy controls automatic retries: MaxAttempts, exponential backoff and the set of retryable status codes.

Load Balancing

TsgcGRPCLoadBalancer distributes calls across several backends using a Pick First or Round Robin policy, and resolves backend addresses through a DNS name resolver.

Health Checking

TsgcGRPCHealthClient implements the standard grpc.health.v1.Health service, with Check for a one-shot status query and Watch to receive health-status changes as a stream.

Server Reflection

TsgcGRPCReflectionClient uses the gRPC Server Reflection service to list the services exposed by a server and to fetch their file descriptors at runtime.

Interceptors

Interceptors can be chained through the Interceptors property. The library includes TsgcGRPCLoggingInterceptor, TsgcGRPCTimeoutInterceptor and TsgcGRPCMetadataInterceptor.

Telemetry

The MetricsCollector exposes per-method telemetry, including call counts, success rate and latency, useful for monitoring and diagnostics.

Protocol Buffers

gRPC requests and responses are Protocol Buffers (protobuf) encoded. The library includes TsgcProtobufWriter and TsgcProtobufMessage to build and parse messages by field number. The Google Cloud interfaces provide typed request and response classes, so you do not need to encode protobuf by hand when calling those services.

Reference

Properties

Name	Description
Client	The TsgcHTTP2Client instance used as the HTTP/2 transport for all gRPC calls.
ChannelOptions	Channel-level options: compression, content-type, maximum message sizes and HTTP/2 keep-alive.

DefaultMetadata	Metadata (custom headers) added to every call, such as an authorization Bearer token.
Interceptors	Chain of interceptors applied to every call (logging, timeout, metadata).
RetryPolicy	Automatic retry configuration: maximum attempts, exponential backoff and retryable status codes.
ServiceConfig	Per-service configuration, including method defaults, retry settings and load-balancing policy.
MetricsCollector	Collects per-method telemetry: call counts, success rate and latency.

Methods

Name	Description
Call	Performs a synchronous unary call and returns the TsgcGRPCResponse.
CallAsync	Performs an asynchronous unary call; the response is delivered through the OnGRPCResponse event.
ServerStreamingCall	Starts a server-streaming call; each server message arrives in OnGRPCStreamMessage.
OpenClientStream	Opens a client-streaming call so the client can send a sequence of messages.
SendStreamMessage	Sends one message on an open client stream.
CloseClientStream	Closes a client stream synchronously and returns the server response.
CloseClientStreamAsync	Closes a client stream asynchronously; the response arrives through OnGRPCResponse.
OpenBidiStream	Opens a bidirectional streaming call where both sides exchange messages.
SendBidiMessage	Sends one message on an open bidirectional stream.
CloseBidiStream	Closes a bidirectional stream and signals end-of-stream to the server.
CancelCall	Cancels an active call or stream before it completes.
CallString	Unary helper that sends a request and returns the response as a string.

Events

Name	Description
OnGRPCConnect	Raised when the underlying HTTP/2 connection to the gRPC server is established.
OnGRPCDisconnect	Raised when the connection to the gRPC server is closed.
OnGRPCResponse	Raised when a unary call completes, delivering the TsgcGRPCResponse.
OnGRPCStreamMessage	Raised for every message received on a server-streaming or bidirectional stream.
OnGRPCStreamEnd	Raised when a stream completes, carrying the final gRPC status.
OnGRPCError	Raised when a call finishes with a non-OK gRPC status code.
OnGRPCException	Raised when an exception occurs while processing a call.

OnGRPCBeforeRequest	Raised before a request is sent, allowing the headers and metadata to be inspected or modified.

See Also

- Google Cloud gRPC services
- TsgcHTTP2Client

Google Cloud gRPC Services

These are typed interfaces built on top of TsgcGRPCClient for calling Google Cloud gRPC APIs. Each interface exposes strongly-typed request and response classes for its service, so you do not need to build or parse Protocol Buffers messages by hand.

Authentication

These interfaces authenticate with a Google service-account JSON key using a self-signed JWT (gcaJWT). The JWT audience, set in GoogleCloudOptions.JWT.API_Endpoint, must match the target service endpoint (for example https://speech.googleapis.com/). The signed token is then passed to the service as an "authorization: Bearer" metadata header on every call.

Services

- **PubSub:** Pub/Sub messaging.
- **Speech:** Speech-to-Text.
- **Translation:** Translation v3.
- **Vision:** Vision image annotation.
- **Natural Language:** Natural Language analysis.
- **Cloud Storage:** Cloud Storage v2.
- **BigQuery:** BigQuery Storage Read.
- **Vertex AI:** Vertex AI generative models.

Each Google API must be enabled on the GCP project before it can be called.

Google Cloud Pub/Sub

Typed interface to call the Google Cloud Pub/Sub gRPC API to publish messages and manage topics and subscriptions.

Introduction

Google Cloud Pub/Sub is a fully managed messaging service for sending and receiving messages between independent applications. The gRPC API is exposed through the **google.pubsub.v1.Publisher** service (topics and publishing) and the **google.pubsub.v1.Subscriber** service (subscriptions and message delivery), reached at **pubsub.googleapis.com:443** over TLS.
Requests and responses are built with typed classes such as **TsgcGRPCPubSubListTopicsRequest** / **TsgcGRPCPubSubListTopicsResponse**, **TsgcGRPCPubSubTopic**, **TsgcGRPCPubSubPublishRequest** / **TsgcGRPCPubSubPublishResponse**, **TsgcGRPCPubSubSubscription**, **TsgcGRPCPubSubPullRequest** / **TsgcGRPCPubSubPullResponse** and **TsgcGRPCPubSubAcknowledgeRequest**, so you do not need to encode Protocol Buffers by hand.
The example below authenticates with a service-account JWT, wires a **TsgcGRPCClient** over a **TsgcHTTP2Client** to the Pub/Sub host, sets the authorization Bearer metadata and calls **ListTopics** for a project:

```
TsgcHTTP2Client *oHTTP2 = new TsgcHTTP2Client();
oHTTP2->Host = "pubsub.googleapis.com";
oHTTP2->Port = 443;
oHTTP2->TLS = true;

TsgcGRPCClient *oGRPC = new TsgcGRPCClient();
oGRPC->Client = oHTTP2;

// service-account JWT authentication
oGRPC->GoogleCloudOptions->JWT->KeyFile = "service-account.json";
oGRPC->GoogleCloudOptions->JWT->API_Endpoint = "https://pubsub.googleapis.com/";
oGRPC->DefaultMetadata->AddValue("authorization", "Bearer " + oGRPC->GoogleCloudOptions->JWT->Token);

// build the typed request and call the method
TsgcGRPCPubSubListTopicsRequest *oRequest = new TsgcGRPCPubSubListTopicsRequest();
oRequest->Project = "projects/my-project-id";
TsgcGRPCResponse *oResponse = oGRPC->Call("google.pubsub.v1.Publisher", "ListTopics", oRequest->ToBytes());
ShowMessage(oResponse->DataString);
delete oRequest;
```

Methods

Name	Description
ListTopics	Publisher service: lists the topics that exist in a project.
CreateTopic	Publisher service: creates a new topic.
DeleteTopic	Publisher service: deletes an existing topic.
Publish	Publisher service: publishes one or more messages to a topic.
CreateSubscription	Subscriber service: creates a subscription on a topic.
DeleteSubscription	Subscriber service: deletes an existing subscription.
Pull	Subscriber service: pulls messages from a subscription.
Acknowledge	Subscriber service: acknowledges messages received from a subscription.
StreamingPull	Subscriber service: bidirectional streaming delivery of messages with flow control.

Demo

A working sample is available in the demo folder **Demos/21.GRPC/10.PubSub**, which shows how to authenticate, list and create topics, publish messages and pull from a subscription.

See Also

- TsgcGRPCClient
- Google Cloud gRPC Services

Google Cloud Speech-to-Text

Typed interface to call the Google Cloud Speech-to-Text gRPC API and transcribe audio into text.

Introduction

Google Cloud Speech-to-Text converts audio into text using Google's speech-recognition models. The gRPC API is exposed through the **google.cloud.speech.v1.Speech** service, reached at **speech.googleapis.com:443** over TLS, and the main method is **Recognize** for synchronous recognition.
Requests are built with **TsgcGRPCSpeechRecognizeRequest**, where **Config** sets Encoding, SampleRateHertz, LanguageCode and EnableAutomaticPunctuation, and **Audio.Uri** points to the audio. The reply is returned in **TsgcGRPCSpeechRecognizeResponse**, whose Results contain Alternatives with Transcript and Confidence.
The example below authenticates with a service-account JWT, wires a **TsgcGRPCClient** over a **TsgcHTTP2Client** to the Speech host, sets the authorization Bearer metadata and calls **Recognize** on a LINEAR16 16000 Hz en-US audio stored at a gs:// URI:

```
TsgcHTTP2Client *oHTTP2 = new TsgcHTTP2Client();
oHTTP2->Host = "speech.googleapis.com";
oHTTP2->Port = 443;
oHTTP2->TLS = true;

TsgcGRPCClient *oGRPC = new TsgcGRPCClient();
oGRPC->Client = oHTTP2;

// service-account JWT authentication
oGRPC->GoogleCloudOptions->JWT->KeyFile = "service-account.json";
oGRPC->GoogleCloudOptions->JWT->API_Endpoint = "https://speech.googleapis.com/";
oGRPC->DefaultMetadata->AddValue("authorization", "Bearer " + oGRPC->GoogleCloudOptions->JWT->Token);

// build the typed request and call the method
TsgcGRPCSpeechRecognizeRequest *oRequest = new TsgcGRPCSpeechRecognizeRequest();
oRequest->Config->Encoding = "LINEAR16";
oRequest->Config->SampleRateHertz = 16000;
oRequest->Config->LanguageCode = "en-US";
oRequest->Config->EnableAutomaticPunctuation = true;
oRequest->Audio->Uri = "gs://my-bucket/audio.wav";
TsgcGRPCResponse *oResponse = oGRPC->Call("google.cloud.speech.v1.Speech", "Recognize", oRequest->ToBytes());
ShowMessage(oResponse->DataString);
delete oRequest;
```

Methods

Name	Description
Recognize	Performs synchronous speech recognition: sends the audio config and audio data, and returns the transcribed text with confidence scores.

Demo

A working sample is available in the demo folder **Demos/21.GRPC/11.Speech_to_Text**, which shows how to authenticate and transcribe an audio file with the Recognize method.

See Also

- TsgcGRPCClient
- Google Cloud gRPC Services

Google Cloud Translation

Typed interface to call the Google Cloud Translation v3 gRPC API to translate and detect the language of text.

Introduction

Google Cloud Translation translates text between languages and can detect the source language automatically. The gRPC API is exposed through the **google.cloud.translation.v3.TranslationService** service, reached at **translate.googleapis.com:443** over TLS, with the main methods **TranslateText**, **DetectLanguage** and **GetSupportedLanguages**.
Requests are built with **TsgcGRPCTranslationTranslateTextRequest**, which sets Parent (projects/<id>/locations/global), Contents, SourceLanguageCode and TargetLanguageCode. The reply is returned in **TsgcGRPCTranslationTranslateTextResponse**, whose Translations contain the TranslatedText.
The example below authenticates with a service-account JWT, wires a **TsgcGRPCClient** over a **TsgcHTTP2Client** to the Translation host, sets the authorization Bearer metadata and calls **TranslateText** to translate "Hello, world!" from en to es:

```
TsgcHTTP2Client *oHTTP2 = new TsgcHTTP2Client();
oHTTP2->Host = "translate.googleapis.com";
oHTTP2->Port = 443;
oHTTP2->TLS = true;

TsgcGRPCClient *oGRPC = new TsgcGRPCClient();
oGRPC->Client = oHTTP2;

// service-account JWT authentication
oGRPC->GoogleCloudOptions->JWT->KeyFile = "service-account.json";
oGRPC->GoogleCloudOptions->JWT->API_Endpoint = "https://translate.googleapis.com/";
oGRPC->DefaultMetadata->AddValue("authorization", "Bearer " + oGRPC->GoogleCloudOptions->JWT->Token);

// build the typed request and call the method
TsgcGRPCTranslationTranslateTextRequest *oRequest = new TsgcGRPCTranslationTranslateTextRequest();
oRequest->Parent = "projects/my-project-id/locations/global";
oRequest->Contents->Add("Hello, world!");
oRequest->SourceLanguageCode = "en";
oRequest->TargetLanguageCode = "es";
TsgcGRPCResponse *oResponse = oGRPC->Call("google.cloud.translation.v3.TranslationService", "TranslateText",
ShowMessage(oResponse->DataString);
delete oRequest;
```

Methods

Name	Description
TranslateText	Translates one or more text strings from a source language to a target language.
DetectLanguage	Detects the language of the supplied text.
GetSupportedLanguages	Returns the list of languages supported for translation.

Demo

A working sample is available in the demo folder **Demos/21.GRPC/12.Translation**, which shows how to authenticate and translate text with the TranslateText method.

See Also

- TsgcGRPCClient
- Google Cloud gRPC Services

Google Cloud Vision

Typed interface to call the Google Cloud Vision gRPC API and analyze images (label, text, face and more).

Introduction

Google Cloud Vision analyzes images and returns insights such as labels, text, faces and landmarks. The gRPC API is exposed through the **google.cloud.vision.v1.ImageAnnotator** service, reached at **vision.googleapis.com:443** over TLS, and the main method is **BatchAnnotateImages**, which annotates a batch of images in a single call.
Requests are built with **TsgcGRPCVisionBatchAnnotateImagesRequest**: call AddRequest to add an image (Image.Source.GcsImageUri for a gs:// image) and AddFeature to request a feature, setting FeatureType (for example LABEL_DETECTION) and MaxResults.
The example below authenticates with a service-account JWT, wires a **TsgcGRPCClient** over a **TsgcHTTP2Client** to the Vision host, sets the authorization Bearer metadata and calls **BatchAnnotateImages** to run label detection on a gs:// image:

```
TsgcHTTP2Client *oHTTP2 = new TsgcHTTP2Client();
oHTTP2->Host = "vision.googleapis.com";
oHTTP2->Port = 443;
oHTTP2->TLS = true;

TsgcGRPCClient *oGRPC = new TsgcGRPCClient();
oGRPC->Client = oHTTP2;

// service-account JWT authentication
oGRPC->GoogleCloudOptions->JWT->KeyFile = "service-account.json";
oGRPC->GoogleCloudOptions->JWT->API_Endpoint = "https://vision.googleapis.com/";
oGRPC->DefaultMetadata->AddValue("authorization", "Bearer " + oGRPC->GoogleCloudOptions->JWT->Token);

// build the typed request and call the method
TsgcGRPCVisionBatchAnnotateImagesRequest *oRequest = new TsgcGRPCVisionBatchAnnotateImagesRequest();
TsgcGRPCVisionAnnotateImageRequest *oImage = oRequest->AddRequest();
oImage->Image->Source->GcsImageUri = "gs://my-bucket/photo.jpg";
oImage->AddFeature("LABEL_DETECTION", 10);
TsgcGRPCResponse *oResponse = oGRPC->Call("google.cloud.vision.v1.ImageAnnotator", "BatchAnnotateImages", oRe
ShowMessage(oResponse->DataString);
delete oRequest;
```

Methods

Name	Description
BatchAnnotateImages	Runs image detection and annotation on a batch of images, returning the requested features (labels, text, faces, etc.).

Demo

A working sample is available in the demo folder **Demos/21.GRPC/13.Vision**, which shows how to authenticate and run label detection on an image with the BatchAnnotateImages method.

See Also

- TsgcGRPCClient

- Google Cloud gRPC Services

Google Cloud Natural Language

Typed interface to call the Google Cloud Natural Language gRPC API to analyze sentiment, entities and content of text.

Introduction

Google Cloud Natural Language extracts insights from text, including sentiment, entities and content categories. The gRPC API is exposed through the **google.cloud.language.v1.LanguageService** service, reached at **language.googleapis.com:443** over TLS, with the main methods **AnalyzeSentiment**, **AnalyzeEntities** and **ClassifyText**.
Requests are built with typed classes such as **TsgcGRPCLanguageAnalyzeSentimentRequest** and **TsgcGRPCLanguageAnalyzeEntitiesRequest**, each carrying a Document whose DocType is set to PLAIN_TEXT together with the text content to analyze.
The example below authenticates with a service-account JWT, wires a **TsgcGRPCClient** over a **TsgcHTTP2Client** to the Natural Language host, sets the authorization Bearer metadata and calls **AnalyzeSentiment** on a plain-text sentence:

```
TsgcHTTP2Client *oHTTP2 = new TsgcHTTP2Client();
oHTTP2->Host = "language.googleapis.com";
oHTTP2->Port = 443;
oHTTP2->TLS = true;

TsgcGRPCClient *oGRPC = new TsgcGRPCClient();
oGRPC->Client = oHTTP2;

// service-account JWT authentication
oGRPC->GoogleCloudOptions->JWT->KeyFile = "service-account.json";
oGRPC->GoogleCloudOptions->JWT->API_Endpoint = "https://language.googleapis.com/";
oGRPC->DefaultMetadata->AddValue("authorization", "Bearer " + oGRPC->GoogleCloudOptions->JWT->Token);

// build the typed request and call the method
TsgcGRPCLanguageAnalyzeSentimentRequest *oRequest = new TsgcGRPCLanguageAnalyzeSentimentRequest();
oRequest->Document->DocType = "PLAIN_TEXT";
oRequest->Document->Content = "I really love this product, it works great!";
TsgcGRPCResponse *oResponse = oGRPC->Call("google.cloud.language.v1.LanguageService", "AnalyzeSentiment", oRe
ShowMessage(oResponse->DataString);
delete oRequest;
```

Methods

Name	Description
AnalyzeSentiment	Analyzes the overall sentiment of a document, returning a score and magnitude.
AnalyzeEntities	Detects named entities (people, places, organizations) in a document.
ClassifyText	Classifies a document into content categories.

Demo

A working sample is available in the demo folder **Demos/21.GRPC/14.Natural_Language**, which shows how to authenticate and analyze the sentiment of a text with the AnalyzeSentiment method.

See Also

- TsgcGRPCClient
- Google Cloud gRPC Services

Google Cloud Storage

Typed interface to call the Google Cloud Storage v2 gRPC API to manage buckets and list objects.

Introduction

Google Cloud Storage is an object-storage service for storing and retrieving any amount of data. The gRPC API is exposed through the **google.storage.v2.Storage** service, reached at **storage.googleapis.com:443** over TLS, with the main methods **ListBuckets**, **CreateBucket**, **DeleteBucket** and **ListObjects**.
Requests are built with typed classes such as **TsgcGRPCStorageListBucketsRequest** (Parent = projects/<id>) and the reply is returned in **TsgcGRPCStorageListBucketsResponse**, whose Buckets contain the bucket Name. Note that the gRPC Storage API also expects an **x-goog-request-params** routing metadata header that identifies the targeted resource; the typed interface adds it for you on each call.
The example below authenticates with a service-account JWT, wires a **TsgcGRPCClient** over a **TsgcHTTP2Client** to the Storage host, sets the authorization Bearer metadata and calls **ListBuckets** for a project:

```
TsgcHTTP2Client *oHTTP2 = new TsgcHTTP2Client();
oHTTP2->Host = "storage.googleapis.com";
oHTTP2->Port = 443;
oHTTP2->TLS = true;

TsgcGRPCClient *oGRPC = new TsgcGRPCClient();
oGRPC->Client = oHTTP2;

// service-account JWT authentication
oGRPC->GoogleCloudOptions->JWT->KeyFile = "service-account.json";
oGRPC->GoogleCloudOptions->JWT->API_Endpoint = "https://storage.googleapis.com/";
oGRPC->DefaultMetadata->AddValue("authorization", "Bearer " + oGRPC->GoogleCloudOptions->JWT->Token);

// build the typed request and call the method
TsgcGRPCStorageListBucketsRequest *oRequest = new TsgcGRPCStorageListBucketsRequest();
oRequest->Parent = "projects/my-project-id";
TsgcGRPCResponse *oResponse = oGRPC->Call("google.storage.v2.Storage", "ListBuckets", oRequest->ToBytes());
ShowMessage(oResponse->DataString);
delete oRequest;
```

Methods

Name	Description
ListBuckets	Lists the buckets that belong to a project.
CreateBucket	Creates a new bucket in a project.
DeleteBucket	Deletes an existing bucket.
ListObjects	Lists the objects stored inside a bucket.

Demo

A working sample is available in the demo folder **Demos/21.GRPC/15.Cloud_Storage**, which shows how to authenticate and list the buckets of a project with the ListBuckets method.

See Also

- TsgcGRPCClient
- Google Cloud gRPC Services

Google Cloud BigQuery Storage

Typed interface to call the Google Cloud BigQuery Storage Read gRPC API to read table data with high throughput.

Introduction

The BigQuery Storage Read API provides fast, high-throughput access to BigQuery table data. The gRPC API is exposed through the **google.cloud.bigquery.storage.v1.BigQueryRead** service, reached at **bigquerystorage.googleapis.com:443** over TLS, with the main methods **CreateReadSession** and **ReadRows** (server streaming).
Requests are built with **TsgcGRPCBigQueryCreateReadSessionRequest**, which sets Parent (projects/<id>), ReadSession.Table (projects/<id>/datasets/<ds>/tables/<tbl>), ReadSession.DataFormat (1 = AVRO, 2 = ARROW) and MaxStreamCount. The session response carries the stream names, which are then read with the server-streaming ReadRows method.
The example below authenticates with a service-account JWT, wires a **TsgcGRPCClient** over a **TsgcHTTP2Client** to the BigQuery Storage host, sets the authorization Bearer metadata and calls **CreateReadSession** on a table using the AVRO data format:

```
TsgcHTTP2Client *oHTTP2 = new TsgcHTTP2Client();
oHTTP2->Host = "bigquerystorage.googleapis.com";
oHTTP2->Port = 443;
oHTTP2->TLS = true;

TsgcGRPCClient *oGRPC = new TsgcGRPCClient();
oGRPC->Client = oHTTP2;

// service-account JWT authentication
oGRPC->GoogleCloudOptions->JWT->KeyFile = "service-account.json";
oGRPC->GoogleCloudOptions->JWT->API_Endpoint = "https://bigquerystorage.googleapis.com/";
oGRPC->DefaultMetadata->AddValue("authorization", "Bearer " + oGRPC->GoogleCloudOptions->JWT->Token);

// build the typed request and call the method
TsgcGRPCBigQueryCreateReadSessionRequest *oRequest = new TsgcGRPCBigQueryCreateReadSessionRequest();
oRequest->Parent = "projects/my-project-id";
oRequest->ReadSession->Table = "projects/my-project-id/datasets/my_dataset/tables/my_table";
oRequest->ReadSession->DataFormat = 1; // 1 = AVRO, 2 = ARROW
oRequest->MaxStreamCount = 1;
TsgcGRPCResponse *oResponse = oGRPC->Call("google.cloud.bigquery.storage.v1.BigQueryRead", "CreateReadSessior
ShowMessage(oResponse->DataString);
delete oRequest;
```

Methods

Name	Description
CreateReadSession	Creates a read session over a table and returns the streams that can be read in parallel.
ReadRows	Server-streaming method that reads the rows of a stream from a read session.

Demo

A working sample is available in the demo folder **Demos/21.GRPC/16.BigQuery**, which shows how to authenticate, create a read session and stream the rows of a table.

See Also

- TsgcGRPCClient
- Google Cloud gRPC Services

Google Vertex AI

Typed interface to call the Google Vertex AI gRPC API to run generative models such as Gemini.

Introduction

Google Vertex AI runs Google's foundation and generative models, including the Gemini family. The gRPC API is exposed through the **google.cloud.aiplatform.v1.PredictionService** service, with the main methods **GenerateContent** and **Predict**.
Vertex AI uses a **regional** endpoint: the host is <region>-aiplatform.googleapis.com (for example us-central1-aiplatform.googleapis.com) on port 443 over TLS, and the JWT audience must match that same host (for example https://us-central1-aiplatform.googleapis.com/).
Requests are built with **TsgcGRPCVertexAIGenerateContentRequest**, which sets the fully qualified Model (projects/<id>/locations/<region>/publishers/google/models/<model>, for example gemini-1.5-flash) together with the prompt content.
The example below authenticates with a service-account JWT, wires a **TsgcGRPCClient** over a **TsgcHTTP2Client** to the regional Vertex AI host, sets the authorization Bearer metadata and calls **GenerateContent** with a prompt against a Gemini model in us-central1:

```
TsgcHTTP2Client *oHTTP2 = new TsgcHTTP2Client();
oHTTP2->Host = "us-central1-aiplatform.googleapis.com";
oHTTP2->Port = 443;
oHTTP2->TLS = true;

TsgcGRPCClient *oGRPC = new TsgcGRPCClient();
oGRPC->Client = oHTTP2;

// service-account JWT authentication (audience must match the regional host)
oGRPC->GoogleCloudOptions->JWT->KeyFile = "service-account.json";
oGRPC->GoogleCloudOptions->JWT->API_Endpoint = "https://us-central1-aiplatform.googleapis.com/";
oGRPC->DefaultMetadata->AddValue("authorization", "Bearer " + oGRPC->GoogleCloudOptions->JWT->Token);

// build the typed request and call the method
TsgcGRPCVertexAIGenerateContentRequest *oRequest = new TsgcGRPCVertexAIGenerateContentRequest();
oRequest->Model = "projects/my-project-id/locations/us-central1/publishers/google/models/gemini-1.5-flash";
oRequest->AddText("Write a haiku about the sea.");
TsgcGRPCResponse *oResponse = oGRPC->Call("google.cloud.aiplatform.v1.PredictionService", "GenerateContent",
ShowMessage(oResponse->DataString);
delete oRequest;
```

Methods

Name	Description
GenerateContent	Generates content from a generative model (such as Gemini) given a prompt.
Predict	Runs a prediction against a deployed model.

Demo

A working sample is available in the demo folder **Demos/21.GRPC/17.Vertex_AI**, which shows how to authenticate against a regional endpoint and generate content from a Gemini model with the GenerateContent method.

See Also

- TsgcGRPCClient
- Google Cloud gRPC Services

HTTP/1

TsgcHTTP1Client is a non-visual component that inherits from TIdHTTP indy component and adds some new properties.
This component is located in sgcHTTP unit.

TLSOptions

Allows you to configure how to connect to secure SSL/TLS servers using the HTTP/1 protocol.

ALPNProtocols: list of the ALPN protocols which will be sent to the server.
RootCertFile: path to root certificate file.
CertFile: path to certificate file.
KeyFile: path to certificate key file.
Password: if the certificate is secured with a password, set it here.
VerifyCertificate: if the certificate must be verified, enable this property. Use the event **OnSSLVerifyPeer** to customize the SSL verification.
VerifyDepth: is an Integer property that represents the maximum number of links permitted when verification is performed for the X.509 certificate.
Version: tlsUndefined by default. With **iohOpenSSL** it enables TLS 1.0 to TLS 1.3 and the highest version both sides support is negotiated. TLS 1.3 requires our custom Indy library. With **iohSChannel** Windows chooses among the TLS versions enabled in the system. With **iohNativeTLS** it negotiates TLS 1.3 or TLS 1.2. If the server requires a specific TLS version, it can be selected here.
Preset: applies a set of secure defaults in a single assignment. With **tlspCustom** (the default) every option keeps the value configured manually. Setting **tlspSecureDefaults** enables VerifyCertificate, raises Version to TLS 1.2 only when it was set to TLS 1.0 or TLS 1.1 (tlsUndefined is left as it is, so TLS 1.3 stays available), sets VersionMin of OpenSSL_Options and SChannel_Options to TLS 1.2 when it was lower or undefined, and enables hostname verification of the server certificate.
Proxy: here you can define if you want to connect through a Proxy Server, you can connect to the following proxy servers:
- **pxyHTTP:** HTTP Proxy Server.
- **pxySocks4:** SOCKS4 Proxy Server.
- **pxySocks4A:** SOCKS4A Proxy Server.
- **pxySocks5:** SOCKS5 Proxy Server.

IOHandler: select which library you will use to connect using TLS.
- **iohOpenSSL:** uses OpenSSL library and is the default for Indy components. Requires deploying OpenSSL libraries for win32/win64.
- **iohSChannel:** uses Secure Channel, which is a security protocol implemented by Microsoft for Windows. It does not require deploying OpenSSL libraries. Only works on Windows 32/64 bits.
- **iohNativeTLS:** uses the pure Object Pascal TLS engine of the sgcCrypto pack, which speaks TLS 1.3 and TLS 1.2 and does not require deploying OpenSSL libraries. With this handler **Version** is the lowest version allowed, not an exact version: tlsUndefined and tls1_2 negotiate TLS 1.3 or TLS 1.2, tls1_3 negotiates TLS 1.3 only, and tls1_0 or tls1_1 raise a configuration error on connect. To connect with TLS 1.2 only, name only TLS 1.2 suites in **NativeTLS_Options.CipherSuites**. See Native TLS.

OpenSSL_Options: configuration of the openSSL libraries.
- **APIVersion:** allows defining which OpenSSL API will be used.
  - **oslAPI_1_0:** uses API 1.0 OpenSSL, it's latest supported by Indy
  - **oslAPI_1_1:** uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).
  - **oslAPI_3_0:** uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).
- **LibPath:** here you can configure where the OpenSSL libraries are located
  - **oslpNone:** this is the default. The OpenSSL libraries should be in the same folder as the binary or in a known path.
  - **oslpDefaultFolder:** automatically sets the OpenSSL path where the libraries should be located for all IDE personalities.
  - **oslpCustomFolder:** if this is the option selected, define the full path in the property LibPathCustom.
- **LibPathCustom:** when LibPath = oslpCustomFolder define here the full path where the OpenSSL libraries are located.

UnixSymLinks: enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
oslsSymLinksDefault: by default, symlinks are enabled except under OSX64 (macOS Monterey and later fail when trying to load the library without a version).
oslsSymLinksLoadFirst: load symlinks first, before trying to load the versioned libraries.
oslsSymLinksLoad: load symlinks after trying to load the versioned libraries.
oslsSymLinksDontLoad: don't load the SymLinks.
VersionMin: set here the minimum version that will use the client to connect to a secure server. By default, the value is tlsUndefined which means the minimum version is the same which has been set in the Version property. Example: if you want to set the Client to only connect using TLS 1.2 or TLS 1.3 set the following values.

```
TLSOptions.Version := tls1_3;
TLSOptions.OpenSSL_Options.VersionMin := tls1_2;
```

X509Checks: use this property to enable additional X509 certificate validations:
Mode: select which options will be validated
oslx509chHostName: verifies the hostname certificate.
oslx509chIPAddress: verifies the ip address of the certificate.
HostName: set the hostname if it's different from the request.
IPAddress: set the ip address if it's different from the request.

SChannel_Options: allows you to use a certificate from Windows Certificate Store.
CertHash: is the certificate Hash. You can find the certificate Hash running a dir command in powershell.
CipherList: here you can set which Ciphers will be used (separated by ":"). Example: CALG_AES_256:CALG_AES_128
CertStoreName: the store name where the certificate is stored. Select one of the following:
scsnMY (the default)
scsnCA
scsnRoot
scsnTrust
CertStorePath: the store path where the certificate is stored. Select one of the following:
scspStoreCurrentUser (the default)
scspStoreLocalMachine

Log

If the Log property is enabled, it saves socket messages to a specified log file, useful for debugging.

LogOptions.FileName: full path to the filename.

Authentication

Allows you to authenticate using **OAuth2** or **JWT**.

RetryOptions

Automatically retries failed HTTP requests using exponential backoff with optional jitter. Retries are disabled by default, so the behaviour is unchanged unless **Enabled** is set to true. A request is retried when the response status code is in **StatusCodes** or when a transient network error occurs (connection closed, socket error, connect or read timeout).
Enabled: enables the automatic retry of failed requests. By default false.
MaxRetries: maximum number of retries after the initial request. By default 3.
InitialInterval: delay in milliseconds before the first retry. By default 500.
MaxInterval: maximum delay in milliseconds between retries. By default 30000.
Multiplier: factor applied to the delay after every attempt. By default 2.0, which doubles the delay on each retry: 500, 1000, 2000...
Jitter: random factor between 0 and 1 applied to every delay (0.2 varies each delay by up to 20%), which avoids many clients retrying at exactly the same time. By default 0.2.

HonorRetryAfter: when the server replies with a Retry-After header (as seconds or as an HTTP date, usually with a 429 or 503 response), the client waits the time requested by the server instead of the computed backoff delay. By default true.

StatusCodes: comma-separated list of HTTP status codes that trigger a retry. By default "429,502,503,504".

Example: retry up to 5 times, starting at 1 second between attempts.

```
TsgcHTTP1Client *oHTTP = new TsgcHTTP1Client();
oHTTP->RetryOptions->Enabled = true;
oHTTP->RetryOptions->MaxRetries = 5;
oHTTP->RetryOptions->InitialInterval = 1000;
ShowMessage(oHTTP->Get("https://www.esegece.com"));
```

The AI provider clients (OpenAI, Anthropic, Gemini and the other LLM API clients) use this same retry engine through the RetryOptions property of their Options.

MaxChunkSize, UseExpect100Continue, UseChunkedTransferEncoding

These three options only take effect when the eSeGeCe custom Indy library is linked in, which ships with the Enterprise and All-Access editions. The Standard and Professional editions build against the Indy library that ships with Delphi, which has no hook for any of the three.

MaxChunkSize: caps the size of a single chunk when reading a chunked response. 0 (the default) means unlimited.

UseExpect100Continue: sends an **Expect: 100-continue** header and waits for the server's acknowledgement before sending the request body. False by default.

UseChunkedTransferEncoding: sends the request body using chunked transfer encoding instead of a Content-Length header. False by default.

On the Standard and Professional editions, leaving all three at their default value (0, False, False) is always safe and behaves the same as on Enterprise and All-Access. Setting any of them to a non-default value raises an exception naming the property and stating that it requires the eSeGeCe custom Indy, instead of silently doing nothing.

The AI provider clients (OpenAI, Anthropic, Gemini and the other LLM API clients) expose the same three properties and are subject to the same edition split, since they use this HTTP client internally.

Asynchronous Requests

By default, the HTTP1Client uses blocking requests, so after calling an HTTP request method, the client waits for the response from the server. Alternatively, you can use asynchronous methods to execute these HTTP requests in a secondary thread, avoiding blocking the thread where the request is called. The following asynchronous methods are implemented:

- GetAsync
- PostAsync
- PutAsync
- OptionsAsync
- DeleteAsync

After calling these methods, instead of waiting for the response, the code continues to the next line, and the response can be handled using the event **OnAsyncResponse**.

```
void __fastcall OnAsyncResultEvent(TObject* Sender, const TsgcHTTPAsyncRequest* aRequest, const TIdHTTPResponse*
```

If there is any error while processing the Asynchronous request, the exception will be raised in the event **OnAsyncException**.

Futures

In Delphi 2010 and later, the client also implements future-style asynchronous requests that do not require any event handler:

- GetFuture
- PostFuture

The request runs on a background thread and the returned future resolves to the response body: call **Await** to block until the response arrives, **ThenProc** to consume it in a continuation and **OnError** to handle failures. Calling **Cancel** aborts the underlying HTTP request instead of letting it run to completion.

Examples

Request a **GET** method to HTTPs server and using **TLS 1.2**

```
TsgcHTTP1Client *oHTTP = new TsgcHTTP1Client();
try
{
  oHTTP->TLSOptions->Version = tls1_2;
  ShowMessage(oHTTP->Get("https://www.google.es"));
}
__finally
{
  oHTTP->Free();
}
```

Request a **GET** method to HTTPs server using **openSSL 1.1** and **TLS 1.3**

```
TsgcHTTP1Client *oHTTP = new TsgcHTTP1Client();
try
{
  oHTTP->TLSOptions->OpenSSL_Options->APIVersion = oslAPI_1_1;
  oHTTP->TLSOptions->Version = tls1_3;
  ShowMessage(oHTTP->Get("https://www.google.es"));
}
__finally
{
  oHTTP->Free();
}
```

Request an **Asynchronous POST** method and read the response using the **OnAsyncResultEvent**.

```
void __fastcall OnAsyncExceptionEvent(TObject* Sender, const TsgcThread* aThread, const Exception* E)
{
    Log(E->Message);
}
void __fastcall OnAsyncResultEvent(TObject* Sender, const TsgcHTTPAsyncRequest* aRequest, const TIdHTTPResponse*
{
    if (aResponse->ResponseCode == 200)
        Log("ok", aRequest->Response);
    else
        Log("error", aRequest->Response);
}
TsgcHTTP1Client* oHTTP = new TsgcHTTP1Client(NULL);
oHTTP->OnAsyncResult = OnAsyncResultEvent;
oHTTP->OnAsyncException = OnAsyncExceptionEvent;
TStringStream* oRequest = new TStringStream("body");
TStringStream* oResponse = new TStringStream("");
oHTTP->PostAsync("https://localhost/test", oRequest, oResponse);
```

Request a GET method to HTTPs server using **SChannel for Windows**.

```
TsgcHTTP1Client *oHTTP = new TsgcHTTP1Client();
try
{
  oHTTP->TLSOptions->IOHandler = iohSChannel;
  oHTTP->TLSOptions->Version = tls1_2;
  ShowMessage(oHTTP->Get("https://www.google.es"));
}
__finally
{
  oHTTP->Free();
}
```

Request **SSE** method to get data events

```
TsgcHTTP1Client *oHTTP = new TsgcHTTP1Client();
oHTTP->OnSSEMessage = OnSSEMessageEvent;
oHTTP->GetSSE("https://www.yoursite.com/sse");

void OnSSEMessageEvent(TObject *Sender, const string aMessage, bool &Cancel)
{
  ShowMessage(aMessage);
}
```

Events

OnSSEMessage

The event is called when a new SSE message is received.

OnSSLVerifyPeer

If verify certificate is enabled, in this event you can verify and decide whether to accept the server certificate.

OnSSLGetHandler

This event is raised before the SSL handler is created. You can create your own SSL handler here (it needs to be inherited from TIdServerIOHandlerSSLBase or TIdIOHandlerSSLBase) and set the properties needed.

OnSSLAfterCreateHandler

If no custom SSL object has been created, a default one is created using the OpenSSL handler. You can access the SSL handler properties and modify them if needed.

OnAsyncResult

The event is called after requesting an Async method (using GetAsync, PutAsync... methods). Use the Response parameter to know the result of the request.

OnAsyncException

If there is any error while processing an async request, this event is called with the exception raised.

Server-Sent Events

Server-Sent Events (SSE) is a standard, defined by the WHATWG HTML specification and used by the browser EventSource API, that lets a server push text events to a client over a long-lived HTTP response with the `text/event-stream` content type. Every event is a block of `field: value` lines ended by an empty line. The fields are `event`, `data`, `id` and `retry`.

```
event: price
id: 1042
data: {"symbol":"EURUSD","bid":1.0842}

retry: 5000
```

The client reconnects when the connection drops and sends the id of the last event received in the `Last-Event-ID` header, so the server can resend what the client missed.

Components

- **TsgcSSEClient:** EventSource client with automatic reconnection and Last-Event-ID resume.
- **TsgcSSEParser:** stand-alone text/event-stream parser, for example to read streaming responses of LLM APIs.

TsgcSSEClient

TsgcSSEClient is a Server-Sent Events (EventSource) client that receives a text/event-stream over HTTP and reconnects automatically.

Introduction

TsgcSSEClient connects to a URL that returns a `text/event-stream`, reads the stream on a background thread and fires an event for every message the server pushes. When the stream ends or the network fails, the client reconnects automatically and sends the `Last-Event-ID` header, so the server can resume the stream from the last event received. A `retry:` field sent by the server sets the reconnection delay for the rest of the session. The component is available in all editions and is registered in the **SGC HTTP** palette tab.

Follow the steps below to configure this component:

1. Create a new instance of **TsgcSSEClient** component and set the **URL** of the event stream.

2. Handle the **OnEvent** event and call **Open**. Example:

```
#include "sgcHTTP_SSE_Client.hpp"

TsgcSSEClient *oSSE = new TsgcSSEClient(NULL);
oSSE->URL = "https://www.example.com/events";
oSSE->OnEvent = OnSSEEvent;
oSSE->Open();

void __fastcall TForm1::OnSSEEvent(TObject *Sender, TsgcSSEEvent* const aEvent)
{
  ShowMessage(aEvent->EventType + ": " + aEvent->Data);
}
```

How the client reacts to the server response:

- **200 with text/event-stream**: the stream opens, OnOpen fires and every event fires OnEvent.
- **3xx redirect**: the redirect is followed.
- **204 No Content**: the server asks the client to stop. The client closes without an error and does not reconnect.
- **Any other status, or a wrong content type**: OnError fires, then OnClose, and the client does not reconnect.
- **Network error or read timeout**: OnError fires and the client reconnects using ReconnectOptions.
- **End of the stream**: the client reconnects using ReconnectOptions.

ReconnectOptions.MaxAttempts counts consecutive failures, and the counter restarts every time a connection opens. Use **Headers** to add request headers such as a bearer token, and **OnBeforeConnect** to configure TLS, a proxy or authentication on the underlying TsgcHTTP1Client.

Reference

- Properties — 6 properties available on the component.
- Methods — 2 public methods.
- Events — 6 events you can handle.

Guides

- **Guides**
  - Resume with Last-Event-ID
- **Classes**
  - TsgcSSEEvent
  - TsgcSSEParser

TsgcSSEClient | Resume with Last-Event-ID

When a server gives its events an `id:` field, TsgcSSEClient stores the last one in the **LastEventId** property and sends it in the `Last-Event-ID` request header every time it reconnects. The server reads that header and sends only the events the client has not received yet, so a dropped connection does not lose messages.

Inside one session this works without any code. To resume after the application restarts, save LastEventId when the client stops and restore it before calling Open. LastEventId keeps its value after Close, and a value set before Open is sent with the first request too.

Restore the id and open the stream

```
oSSE->URL = "https://www.example.com/events";
oSSE->OnClose = OnSSEClose;
if (FileExists("sse_last_id.txt"))
{
  TStringList *oFile = new TStringList();
  try
  {
    oFile->LoadFromFile("sse_last_id.txt");
    oSSE->LastEventId = Trim(oFile->Text);
  }
  __finally
  {
    delete oFile;
  }
}
oSSE->Open();
```

Close the stream and save the id

```
oSSE->Close();
TStringList *oFile = new TStringList();
try
{
  oFile->Text = oSSE->LastEventId;
  oFile->SaveToFile("sse_last_id.txt");
}
__finally
{
  delete oFile;
}
```

Resuming only works when the server supports it: it must send an `id:` field with the events and read the `Last-Event-ID` header of the request.

TsgcSSEEvent

Event received from a text/event-stream. It is passed to the OnEvent event of TsgcSSEClient and of TsgcSSEParser.

Name	Type	Description
`EventType`	`string`	Value of the `event:` field, or `message` when the server does not send one.
`Data`	`string`	Value of the `data:` lines. When an event has several data lines, they are joined with a line feed.
`Id`	`string`	Last event id known when the event was dispatched.

The object belongs to the client or the parser and is only valid inside the event handler. Copy the values you need to keep.

TsgcSSEParser

TsgcSSEParser is the text/event-stream interpreter used by TsgcSSEClient, and it can be used on its own. Feed it the raw bytes or the text of a stream in chunks of any size. Every complete event fires OnEvent and every valid `retry:` field fires OnRetry. It is useful to parse streaming responses of HTTP APIs that use the SSE format, for example the streamed answers of LLM providers, when the stream is read with another HTTP client. It is declared in the unit `sgcHTTP_SSE_Client`.

Member	Description
`procedure Feed(const aBytes: TBytes); overload;`	Parses a chunk of raw UTF-8 bytes. A chunk can end in the middle of a line or of an event, the rest is kept until the next call.
`procedure Feed(const aText: string); overload;`	Parses a chunk of text.
`procedure Reset;`	Discards the pending line and event, to start parsing a new stream.
`property LastEventId: string;`	Last event id read from an `id:` field. It can be set to start with a known value.
`property OnEvent: TsgcSSEEventEvent;`	Fires for every complete event with a TsgcSSEEvent.
`property OnRetry: TsgcSSERetryEvent;`	Fires for every valid `retry:` field with the value in milliseconds. `TsgcSSERetryEvent = procedure(Sender: TObject; aRetry: Integer) of object`.

Example

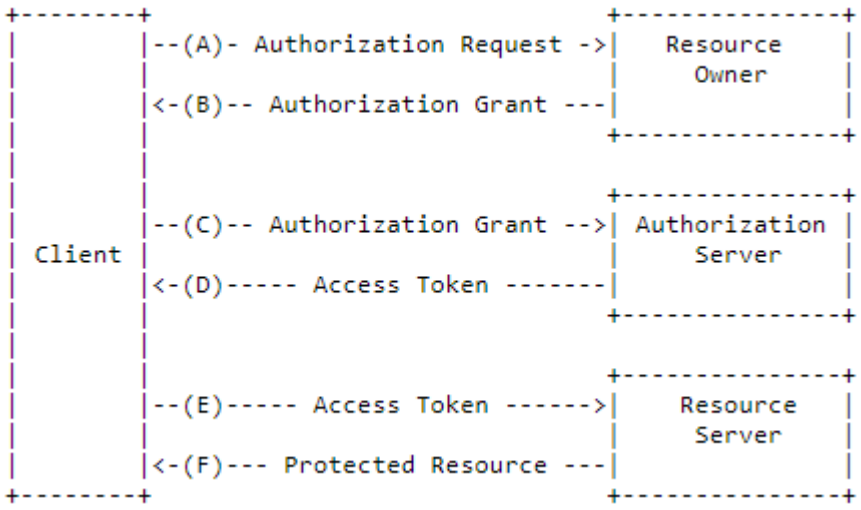
```
TsgcSSEParser *oParser = new TsgcSSEParser();
try
{
  oParser->OnEvent = OnParserEvent;
  oParser->OnRetry = OnParserRetry;
  // chunks can split a line or an event anywhere
  oParser->Feed(String("event: delta\ndata: {\"text\":\"Hel"));
  oParser->Feed(String("lo\"}\n\n"));
  oParser->Feed(TEncoding::UTF8->GetBytes(String("id: 7\ndata: done\n\n")));
  ShowMessage("last id: " + oParser->LastEventId);
  // start again for a new response
  oParser->Reset();
}
__finally
{
  delete oParser;
}

void __fastcall TForm1::OnParserEvent(TObject *Sender, TsgcSSEEvent* const aEvent)
{
  ShowMessage(aEvent->EventType + " " + aEvent->Id + " " + aEvent->Data);
}

void __fastcall TForm1::OnParserRetry(TObject *Sender, int aRetry)
{
  ShowMessage("server asks to retry after " + IntToStr(aRetry) + " ms");
}
```

HTTP | OAuth2

OAuth2 allows third-party applications to receive limited access to an HTTP service, either on behalf of a resource owner or by allowing a third-party application to obtain access on its own behalf. Thanks to OAuth2, service providers and consumer applications can interact with each other in a secure way.

In OAuth2, there are 4 roles:

- **Resource Owner:** the user.
- **Resource Server:** the server that hosts the protected resources and provides access to it based on the access token.
- **Client:** the external application that seeks permission.
- **Authorization Server:** issues the access token after having authenticated the user.



Components

- **TsgcHTTP_OAuth2_Client:** is a client with support for OAuth2, so it can connect to OAuth2 servers to request an authentication like Google, Facebook...
- **TsgcHTTP_OAuth2_Server:** is the server implementation of OAuth2 protocol, allows you to protect the resources of the Server.
- **TsgcHTTP_OAuth2_Server_Provider:** allows implementing external OAuth2 Providers (like Azure AD, Google, Facebook...) in your Server, so the user can login using the Azure, Google, Facebook... user credentials.

Server and **Client** OAuth2 components support **PKCE** (Proof Key for Code Exchange), which is an extension to the Authorization Code flow to prevent CSRF and authorization code injection attacks (RFC 7636).

TsgcHTTP_OAuth2_Client

When a client needs a new Access Token, automatically starts an HTTP server to process response from Authorization server.

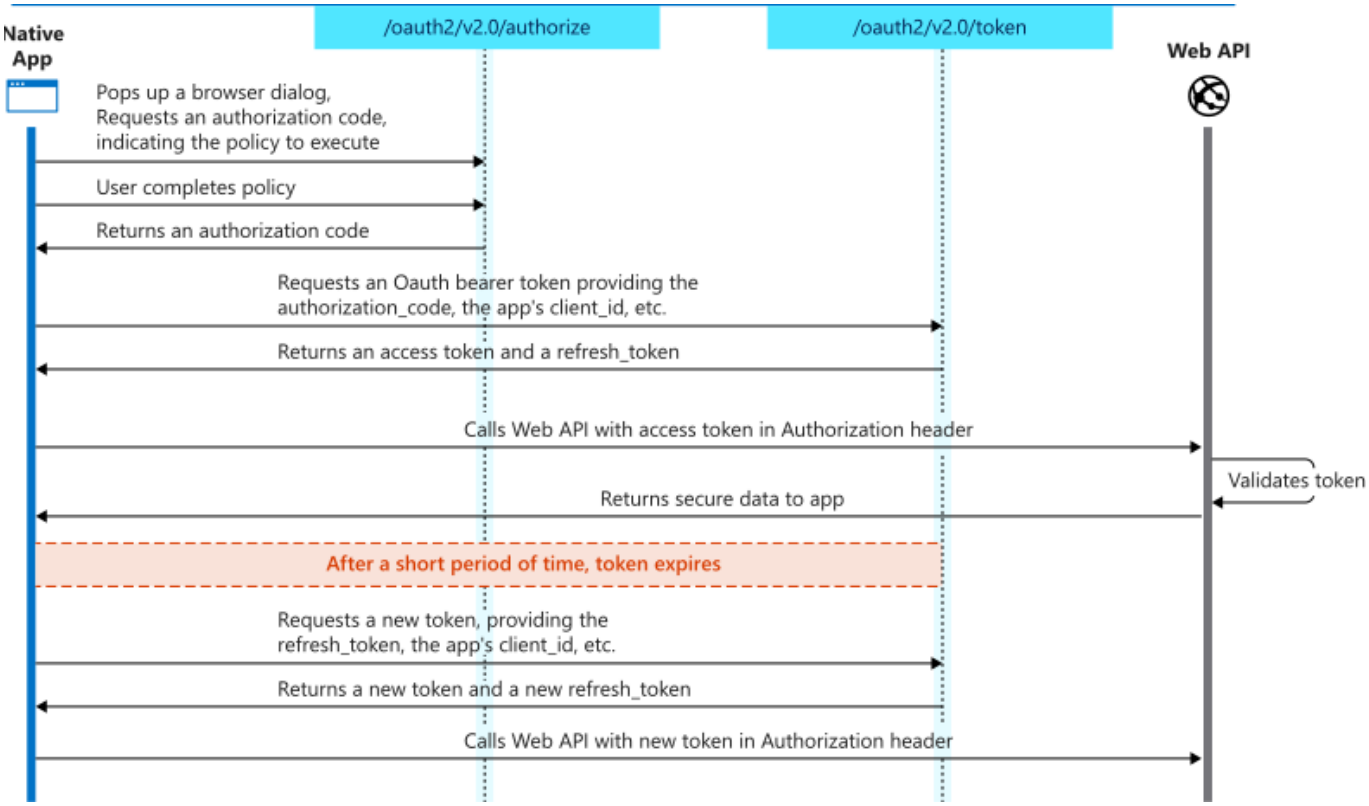
Introduction

This component allows you to handle flow between client and the other roles, basically, when you set Active := True, opens a new Web Browser and requests user grant authorization, if successful, authorization server sends a token to application which is processed and with this token, client can connect to resource server. This component, starts a simple HTTP server which handles authorization server responses and uses an HTTP client to request Access Tokens.

GrantType

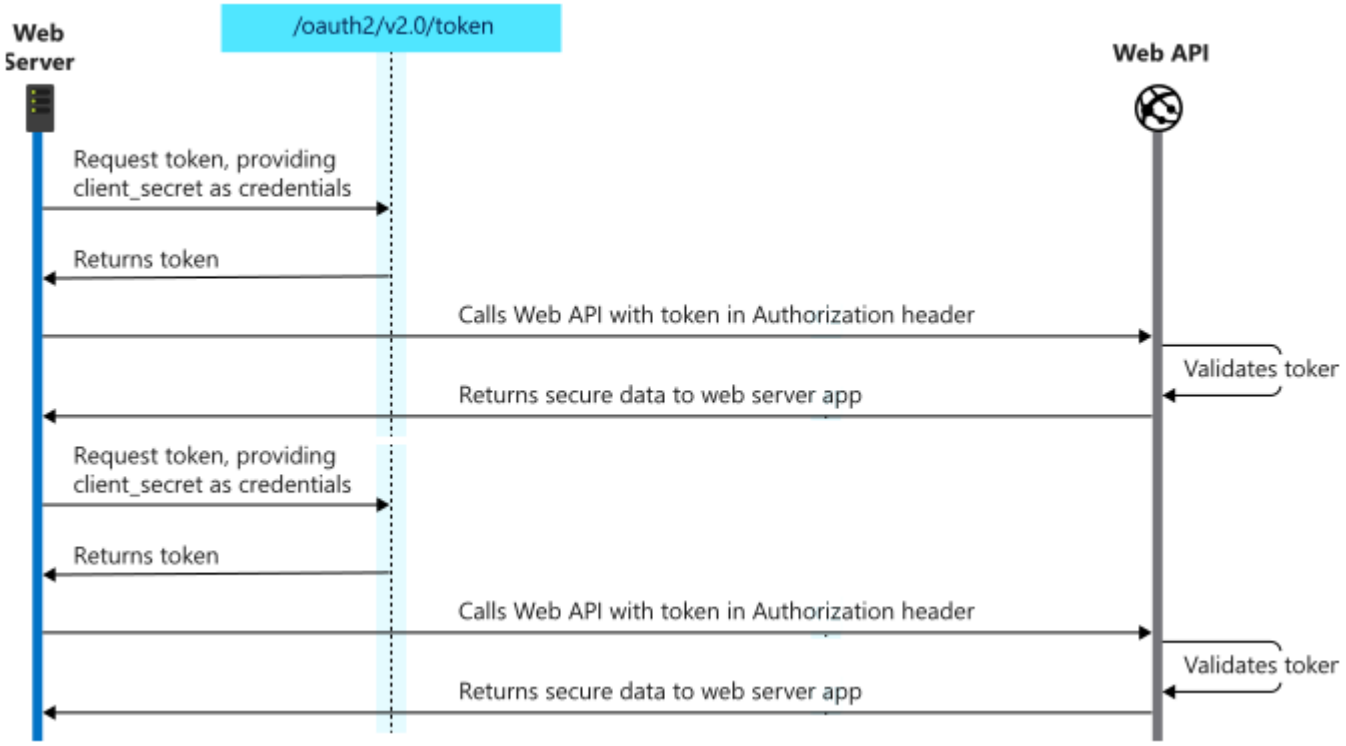
Client supports the following types of Authorization:

auth2Code: It's used to perform authentication and authorization in the majority of application types, including single page applications, web applications, and natively installed applications. The flow enables apps to securely acquire access_tokens that can be used to access resources secured, as well as refresh tokens to get additional access_tokens, and ID tokens for the signed in user.



auth2CodePKCE: it's the same authentication flow than auth2Code with PKCE enabled. PKCE (Proof Key for Code Exchange) is a security extension for OAuth 2.0, designed to enhance the security of authorization flows for native and single-page applications. It mitigates the risk of interception attacks, especially in public clients where the authorization code might be exposed to interception in transit. Usually this option is used in native and mobile apps.

auth2ClientCredentials: This type of grant is commonly used for server-to-server interactions that must run in the background, without immediate interaction with a user. These types of applications are often referred to as daemons or service accounts.



auth2DeviceCode: Device Authorization Grant (RFC 8628) for input-constrained devices such as smart TVs, media consoles, and IoT devices that lack a browser or have limited input capabilities. The device displays a user code and verification URI; the user visits the URI on a secondary device (phone or computer) and enters the code to authorize.

LocalServerOptions

When a client needs a new Access Token, automatically starts an HTTP server to process response from Authorization server. This server is transparent for user and usually works in localhost. By default uses port 8080 but you can change if needed.

- **IP:** IP server listening, example: 127.0.0.1
- **Port:** by default 8080. When using GrantType = auth2CodePKCE (for desktop and native application), you can set the value of the port to zero and the server will choose a random port.
- **RedirectURL:** (optional) allows customizing the redirect URL, example: http://localhost:8080/oauth/.
- **SSL:** enable this property if local server runs on a secure port (*only supported by Professional and Enterprise Editions).
- **SSLOptions:** allows customizing the SSL properties of server (*only supported by Professional and Enterprise Editions).
- **LogOptions:** allows you to save the log of the Requests/Responses received and sent by the HTTP Internal Server (*Only for Professional and Enterprise Editions).
    - **Enabled:** set to True to enable the log to file.
    - **FileName:** set the file name to store the log file.

AuthorizationServerOptions

Here you must set URL for Authorization and Acces Token, usually these are provided in API specification. Scope is a list of all scopes requested by client. Example:

- **AuthURL:** https://accounts.google.com/o/oauth2/auth
- **TokenURL:** https://accounts.google.com/o/oauth2/token
- **Scope:** https://mail.google.com/
- **RevocationURL:** URL for token revocation endpoint (required for Revoke method). Example: https://accounts.google.com/o/oauth2/revoke
- **IntrospectionURL:** URL for token introspection endpoint (required for Introspect method). Example: https://accounts.google.com/o/oauth2/introspect

OAuth2Options

ClientId is a mandatory field which informs server which is the identification of client. Check your API specification to know how get a ClientId. The same applies for client secret.
Sometimes, server requires a user and password to connect using Basic Authentication, if this is the case, you can setup this in Username/Password fields. Example:

- **GrantType:** Authorization flow type
  - **auth2Code:** trusted apps, like a web-server.
  - **auth2CodePKCE:** untrusted native or mobile apps.
  - **auth2ClientCredentials:** automated apps without user interaction.
  - **auth2ResourceOwnerPassword:** allows an application to sign in the user by directly handling their password
  - **auth2DeviceCode:** Device Authorization Grant (RFC 8628) for input-constrained devices without a browser or with limited input capabilities.
- **ClientId:** 180803918307-eqjtm20gqfhcs6gjklbbrreng022mqqc.apps.googleusercontent.com
- **ClientSecret:** _by1iYYrvVHxC2Z8TbtNEYJN
- **Username:**
- **Password:**

HTTPClientOptions

Here you can customize the Client Options when connects to HTTP Server to request a new token.

TLSOptions: if TLS enabled, here you can customize some TLS properties.

ALPNProtocols: list of the ALPN protocols which will be sent to server.
RootCertFile: path to root certificate file.
CertFile: path to certificate file.
KeyFile: path to certificate key file.
Password: if certificate is secured with a password, set here.
VerifyCertificate: if certificate must be verified, enable this property.
VerifyDepth: is an Integer property that represents the maximum number of links permitted when verification is performed for the X.509 certificate.
Version: by default uses TLS 1.0, if server requires a higher TLS version, here can be selected.
IOHandler: select which library you will use to connect using TLS.
iohOpenSSL: uses OpenSSL library and is the default for Indy components. Requires to deploy openssl libraries for win32/win64.
iohSChannel: uses Secure Channel which is a security protocol implemented by Microsoft for Windows, doesn't require to deploy openssl libraries. Only works in Windows 32/64 bits.
OpenSSL_Options: allows defining which OpenSSL API will be used.
APIVersion: allows defining which OpenSSL API will be used.
oslAPI_1_0: uses API 1.0 OpenSSL, it's latest supported by Indy
oslAPI_1_1: uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).
oslAPI_3_0: uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).
LibPath: here you can configure where are located the openSSL libraries
oslpNone: this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.
oslpDefaultFolder: sets automatically the openSSL path where the libraries should be located for all IDE personalities.
oslpCustomFolder: if this is the option selected, define the full path in the property LibPathCustom.
LibPathCustom: when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.
UnixSymLinks: enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
oslsSymLinksDefault: by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).
oslsSymLinksLoadFirst: Load SymLinks and do before trying to load the version libraries.
oslsSymLinksLoad: Load SymLinks after trying to load the version libraries.
oslsSymLinksDontLoad: don't load the SymLinks.
SChannel_Options: allows you to use a certificate from Windows Certificate Store.
CertHash: is the certificate Hash. You can find the certificate Hash running a dir command in powershell.
CertStoreName: the store name where is stored the certificate. Select one of below:

scsnMY (the default)
scsnCA
scsnRoot
scsnTrust

CertStorePath: the store path where is stored the certificate. Select one of below:
scspStoreCurrentUser (the default)
scspStoreLocalMachine

LogOptions: if a filename is set, it will save a log of HTTP requests/responses of the HTTP client

Reference

- Properties — 6 properties available on the component.
- Methods — 7 public methods.
- Events — 19 events you can handle.

OAuth2 Client for Web Applications

When the OAuth2 Client must to get an Authorized Token connecting to a Web Application, the Local Server (used to get the authorized token) must be configured with the Web Application Parameters. Set the following **LocalServerOptions** properties:

- **IP:** Address IP previously configured in the OAuth2 Configuration.
- **Port:** Listening Port previously configured in the OAuth2 Configuration.
- **SSL:** if the server is using a secure connection, enable this option.

OAuth2 Client for Desktop Applications

When the OAuth2 Client is a Desktop or native application, the Local Server (used to get the authorized token) can be listening on a local ip address (example: 127.0.0.1) and the port can be chosen randomly. So, the **LocalServerOptions** should be configured as follows

- **IP:** 127.0.0.1
- **Port:** set to zero, and the server will choose automatically a random port.

When creating an OAuth2 Client for Desktop or Native applications, set the **OAuth2Options.GrantTypе** to **auth2CodePKCE** to add an extra security.

OAuth2 | TsgcHTTP_OAuth2_Client_Google

This component lets you login with your Google Account in an easy way.

Configuration

The module requires first **configure your OAuth2 Application** in your Google Account, once are configure just add a couple of lines in your application to allow users login with any Google Account.

The **Local Server** used to read the response from Google, by default listens on **IP Address 127.0.0.1** and **port 0** (random port). So you must configure the CallBack URL in the Google Application. Of course, you can modify the IP Address and port.

Once configured the OAuth2 Application in the Google Account, just create an instance of **TsgcHTTP_OAuth2_Client_Google** and call the method **Authenticate** passing as parameters the **Client_Id** and **Client_Secret**. This **method waits** (by default up to 60 seconds) till the user has **login success-fully**. Returns an object where you can check if the user has authenticated or not, the Name, Id... and more data from the user profile.

Example

```
void GoogleSignIn()
{
  TsgcHTTP_OAuth2_Client_Google *oClient = new TsgcHTTP_OAuth2_Client_Google(NULL);
  try
  {
    TsgcOAuth2_Google_Data *oData = oClient->Authenticate("client_id", "client_secret");
    if (oData->Authenticated)
      ShowMessage(oData->UserProfile->_Name);
  }
  __finally
  {
    delete oClient;
  }
}
```

TsgcHTTP_OAuth2_Client_Microsoft

This component lets you login with your Microsoft Account in an easy way.

Configuration

The module requires first **configure your OAuth2 Application** in your Microsoft Account, once are configure just add a couple of lines in your application to allow users login with any Microsoft Account.

The **Local Server** used to read the response from Microsoft, by default listens on **IP Address 127.0.0.1** and **port 8080** and uses SSL. So you must configure the CallBack URL as **https://localhost:8080** (Microsoft only allows localhost as a local IP Address) in the Microsoft Application. Of course, you can modify the IP Address and port.

Once configured the OAuth2 Application in the Microsoft Account, just create an instance of **TsgcHTTP_OAuth2_Client_Microsoft** and call the method **Authenticate** passing as parameters the **TenantId**, **Client_Id** This **method waits** (by default up to 60 seconds) till the user has **login successfully**. Returns an object where you can check if the user has authenticated or not, the Name, Id... and more data from the user profile.

Example

```
void MicrosoftSignIn()
{
  TsgcHTTP_OAuth2_Client_Microsoft *oClient = new TsgcHTTP_OAuth2_Client_Microsoft(NULL);
  try
  {
    TsgcOAuth2_Microsoft_Data *oData = oClient->Authenticate("tenant_id", "client_id", "client_secret");
    if (oData->Authenticated)
      ShowMessage(oData->UserProfile->DisplayName);
  }
  __finally
  {
    delete oClient;
  }
}
```

OAuth2 | Authorization Code Grant (RFC 6749)

Overview

The Authorization Code grant is the most common OAuth2 flow. It is designed for web applications, desktop applications, and mobile applications where the client can securely store a client secret. The flow involves redirecting the user to the authorization server, where they authenticate and grant permission. The server then returns an authorization code to a redirect URI, and the client exchanges that code for an access token.

Flow

1. Client redirects the user to the **AuthURL** with client_id, redirect_uri, response_type=code, scope, and state parameters.
2. User authenticates with the authorization server and grants permission to the client.
3. Authorization server redirects the user back to the redirect URI with a `code` parameter.
4. Client POSTs the authorization code to the **TokenURL** along with client_id, client_secret, and redirect_uri to exchange it for an access token.
5. Server returns `access_token`, `token_type`, `expires_in`, and optionally a `refresh_token`.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2Code**.
`OAuth2Options.ClientId`	The client identifier issued by the authorization server.
`OAuth2Options.ClientSecret`	The client secret issued by the authorization server.
`AuthorizationServerOptions.AuthURL`	The authorization endpoint URL where the user is redirected to authenticate.
`AuthorizationServerOptions.TokenURL`	The token endpoint URL where the authorization code is exchanged for an access token.
`AuthorizationServerOptions.Scope`	The scope of the access request (e.g., openid, profile, email).
`LocalServerOptions.IP`	The IP address of the local redirect server (e.g., 127.0.0.1).
`LocalServerOptions.Port`	The port of the local redirect server (e.g., 8080).

Events

Event	Description
`OnBeforeAuthorizeCode`	Fired before the client redirects the user to the authorization endpoint. Allows customization of the authorization request.

`OnAfterAuthorizeCode`	Fired after the authorization code is received from the server.
`OnErrorAuthorizeCode`	Fired when an error occurs during the authorization code step.
`OnBeforeAccessToken`	Fired before the client exchanges the authorization code for an access token.
`OnAfterAccessToken`	Fired after the access token is received. Use this event to retrieve the token value.
`OnErrorAccessToken`	Fired when an error occurs during the token exchange step.

Example

```
OAuth2->OAuth2Options->GrantType = auth2Code;
OAuth2->OAuth2Options->ClientId = "your-client-id";
OAuth2->OAuth2Options->ClientSecret = "your-client-secret";
OAuth2->AuthorizationServerOptions->AuthURL = "https://provider.com/oauth2/authorize";
OAuth2->AuthorizationServerOptions->TokenURL = "https://provider.com/oauth2/token";
OAuth2->AuthorizationServerOptions->Scope->Text = "openid profile";
OAuth2->LocalServerOptions->IP = "127.0.0.1";
OAuth2->LocalServerOptions->Port = 8080;
OAuth2->Start();
```

Token Refresh

When an access token expires, you can use the refresh token to obtain a new one without requiring user interaction. Call the `Refresh` method with the refresh token value:
The `OnAfterAccessToken` event will fire with the new access token. If the refresh fails, the `OnErrorAccessToken` event will fire.

When to Use

Use the Authorization Code grant for:

- Web applications with a server-side backend that can securely store the client secret.
- Server-side rendered applications (e.g., ASP.NET, PHP, Java).
- Desktop applications where the client secret can be reasonably protected.

For public clients (native apps, SPAs) where the client secret cannot be stored securely, consider using Authorization Code with PKCE instead.

OAuth2 | Authorization Code with PKCE (RFC 7636)

Overview

The Authorization Code with PKCE (Proof Key for Code Exchange) grant extends the standard Authorization Code flow with an additional security layer. It is designed for native applications, mobile applications, and single-page applications (SPAs) where the client secret cannot be stored securely. Instead of relying on a client secret, the client generates a cryptographic code verifier and challenge that prove the entity exchanging the authorization code is the same entity that initiated the flow.

How PKCE Works

1. Client generates a random **code_verifier** (32 bytes of cryptographically random data, Base64URL encoded).
2. Client computes the **code_challenge** as the SHA-256 hash of the code_verifier, Base64URL encoded.
3. Client sends the `code_challenge` and `code_challenge_method=S256` with the authorization request.
4. After user authorization, the client sends the original `code_verifier` with the token exchange request.
5. The authorization server verifies that `SHA256(code_verifier) == code_challenge` before issuing the token.

This prevents authorization code interception attacks because an attacker who intercepts the code cannot exchange it without the original code_verifier.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2CodePKCE**.
`OAuth2Options.ClientId`	The client identifier issued by the authorization server.
`OAuth2Options.ClientSecret`	Optional. Some providers require it even with PKCE; others do not.
`AuthorizationServerOptions.AuthURL`	The authorization endpoint URL where the user is redirected to authenticate.
`AuthorizationServerOptions.TokenURL`	The token endpoint URL where the authorization code is exchanged for an access token.
`AuthorizationServerOptions.Scope`	The scope of the access request.
`LocalServerOptions.IP`	The IP address of the local redirect server (e.g., 127.0.0.1).
`LocalServerOptions.Port`	The port of the local redirect server. Set to **0** for a random available port.

Example

```
OAuth2->OAuth2Options->GrantType = auth2CodePKCE;
OAuth2->OAuth2Options->ClientId = "your-client-id";
OAuth2->AuthorizationServerOptions->AuthURL = "https://provider.com/oauth2/authorize";
```

```
OAuth2->AuthorizationServerOptions->TokenURL = "https://provider.com/oauth2/token";
OAuth2->AuthorizationServerOptions->Scope->Text = "openid profile";
OAuth2->LocalServerOptions->IP = "127.0.0.1";
OAuth2->LocalServerOptions->Port = 0;
OAuth2->Start();
```

Random Port

When `LocalServerOptions.Port` is set to **0**, the component automatically selects a random available port. This is recommended for desktop and mobile applications because it avoids port conflicts. The selected port is included in the redirect URI sent to the authorization server.

Token Refresh

PKCE flows typically return a refresh token. Use the `Refresh` method to obtain a new access token:

When to Use

Use the Authorization Code with PKCE grant for:

- Desktop applications (native Windows, macOS, Linux).
- Mobile applications (iOS, Android).
- Single-page applications (SPAs) running in the browser.
- Any public client that cannot securely store a client secret.
- As a more secure alternative to the standard Authorization Code grant, even for confidential clients.

OAuth2 | Client Credentials Grant (RFC 6749)

Overview

The Client Credentials grant is used for machine-to-machine (M2M) authentication where no user interaction is required. The client authenticates directly with the authorization server using its own credentials (client_id and client_secret) and receives an access token. There is no authorization code step and no user login.
This grant type is appropriate when the client is acting on its own behalf rather than on behalf of a user.

Flow

1. Client POSTs `client_id`, `client_secret`, `grant_type=client_credentials`, and `scope` to the token endpoint.
2. Authorization server validates the client credentials.
3. Server returns `access_token`, `token_type`, and `expires_in`.

Note that refresh tokens are typically not issued with this grant type, since the client can simply request a new access token using its credentials.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2ClientCredentials**.
`OAuth2Options.ClientId`	The client identifier issued by the authorization server.
`OAuth2Options.ClientSecret`	The client secret issued by the authorization server.
`AuthorizationServerOptions.TokenURL`	The token endpoint URL. No AuthURL is needed since there is no user authorization step.
`AuthorizationServerOptions.Scope`	The scope of the access request.

Example

```
OAuth2->OAuth2Options->GrantType = auth2ClientCredentials;
OAuth2->OAuth2Options->ClientId = "your-client-id";
OAuth2->OAuth2Options->ClientSecret = "your-client-secret";
OAuth2->AuthorizationServerOptions->TokenURL = "https://provider.com/oauth2/token";
OAuth2->AuthorizationServerOptions->Scope->Text = "api.read api.write";
OAuth2->Start();
```

When to Use

Use the Client Credentials grant for:

- Backend services and daemons that run without user interaction.
- Server-to-server API communication.
- Scheduled jobs, batch processes, and cron tasks.
- Microservice-to-microservice authentication.
- Any scenario where the application acts on its own behalf, not on behalf of a user.

OAuth2 | Resource Owner Password Credentials Grant (RFC 6749)

Overview

The Resource Owner Password Credentials (ROPC) grant allows the client to collect the user's username and password directly and exchange them for an access token. The user's credentials are sent to the token endpoint, and the server returns an access token if they are valid.
This grant type should only be used when there is a high degree of trust between the resource owner and the client, such as when the client is a first-party application developed by the same organization that operates the authorization server.

Security Warning

This grant type is considered less secure than other OAuth2 flows. It exposes the user's credentials directly to the client application, which violates the principle of delegated authorization. The OAuth 2.1 specification has deprecated this grant type. Use Authorization Code with PKCE instead whenever possible.

Flow

1. User provides username and password directly to the client application.
2. Client POSTs `username`, `password`, `client_id`, `client_secret`, `grant_type=password`, and `scope` to the token endpoint.
3. Authorization server validates the credentials.
4. Server returns `access_token`, `token_type`, `expires_in`, and optionally a `refresh_token`.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2ResourceOwnerPassword**.
`OAuth2Options.ClientId`	The client identifier issued by the authorization server.
`OAuth2Options.ClientSecret`	The client secret issued by the authorization server.
`OAuth2Options.UserName`	The resource owner's username.
`OAuth2Options.Password`	The resource owner's password.
`AuthorizationServerOptions.TokenURL`	The token endpoint URL. No AuthURL is needed since user credentials are provided directly.
`AuthorizationServerOptions.Scope`	The scope of the access request.

Example

```
OAuth2->OAuth2Options->GrantType = auth2ResourceOwnerPassword;
OAuth2->OAuth2Options->ClientId = "your-client-id";
OAuth2->OAuth2Options->ClientSecret = "your-client-secret";
```

```
OAuth2->OAuth2Options->UserName = "user@example.com";
OAuth2->OAuth2Options->Password = "user-password";
OAuth2->AuthorizationServerOptions->TokenURL = "https://provider.com/oauth2/token";
OAuth2->AuthorizationServerOptions->Scope->Text = "openid profile";
OAuth2->Start();
```

Token Refresh

If the server returns a refresh token, you can use it to obtain a new access token without re-entering the user's credentials:

When to Use

Use the Resource Owner Password Credentials grant only for:

- Legacy applications that cannot be migrated to a more secure flow.
- First-party applications where you control both the client and the authorization server.
- Migration scenarios where you are transitioning from direct username/password authentication to OAuth2.

For all other cases, use Authorization Code with PKCE or Authorization Code instead.

OAuth2 | Device Authorization Grant (RFC 8628)

Overview

The Device Authorization grant (also known as Device Code flow) is designed for input-constrained devices that cannot easily display a browser or accept keyboard input. Examples include smart TVs, IoT devices, media players, CLI tools, and kiosks. The device displays a short code that the user enters on another device (such as a phone or computer) to complete the authorization.

Flow

1. Client POSTs `client_id` and `scope` to the **DeviceAuthorizationURL**.
2. Server returns `device_code`, `user_code`, `verification_uri`, `expires_in`, and `interval`.
3. Client fires the **OnDeviceCode** event with the user code and verification URI. The application displays these to the user (e.g., "Go to https://provider.com/device and enter code: ABCD-1234").
4. User navigates to the verification URI on another device and enters the user code.
5. User authenticates and grants permission on the secondary device.
6. Meanwhile, the client automatically polls the token endpoint every `interval` seconds using the device code.
7. When the user completes authorization, the next poll returns the `access_token`.
8. If the device code expires before the user authorizes, the **OnDeviceCodeExpired** event fires.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2DeviceCode**.
`OAuth2Options.ClientId`	The client identifier issued by the authorization server.
`OAuth2Options.ClientSecret`	Optional. Some providers require it, others do not.
`AuthorizationServerOptions.TokenURL`	The token endpoint URL used for polling.
`AuthorizationServerOptions.DeviceAuthorizationURL`	The device authorization endpoint where the client requests a device code.
`AuthorizationServerOptions.Scope`	The scope of the access request.

Events

Event	Description
`OnDeviceCode`	Fired when the device code and user code are received from the server. Display the user code and verification URI to the user.

`OnDeviceCodeExpired`	Fired when the device code expires before the user completes authorization. The application should prompt the user to restart the flow.
`OnAfterAccessToken`	Fired when the user completes authorization and the access token is received.
`OnErrorAccessToken`	Fired when a non-recoverable error occurs during the device flow.

Example

Handling the Device Code Event

```
OAuth2->OAuth2Options->GrantType = auth2DeviceCode;
OAuth2->OAuth2Options->ClientId = "your-client-id";
OAuth2->AuthorizationServerOptions->TokenURL = "https://provider.com/oauth2/token";
OAuth2->AuthorizationServerOptions->DeviceAuthorizationURL = "https://provider.com/oauth2/device/code";
OAuth2->AuthorizationServerOptions->Scope->Text = "openid profile";
OAuth2->Start();
```

Polling Behavior

After calling `Start`, the component automatically polls the token endpoint at the interval specified by the server (typically 5 seconds). If the server responds with a `slow_down` error, the component increases the polling interval by 5 seconds. Polling continues until one of the following occurs:

- The user completes authorization and the token is returned.
- The device code expires (`OnDeviceCodeExpired` fires).
- An unrecoverable error occurs (`OnErrorAccessToken` fires).

When to Use

Use the Device Authorization grant for:

- Smart TVs and streaming devices.
- Command-line interface (CLI) tools.
- IoT devices with limited input capability.
- Kiosks and digital signage.
- Any device where typing a full URL or credentials is impractical.

OAuth2 | DPoP - Demonstrating Proof of Possession (RFC 9449)

Overview

DPoP (Demonstrating Proof of Possession) is a security mechanism that binds access tokens to a specific client by requiring proof of possession of a private key. Unlike bearer tokens, which can be used by anyone who obtains them, DPoP-bound tokens are useless without the corresponding private key. This provides protection against token theft and replay attacks.
DPoP is not a grant type itself but a security enhancement that can be used with any OAuth2 grant type (Authorization Code, Client Credentials, etc.).

How It Works

1. Client generates an asymmetric key pair (ES256 or RS256).
2. Client creates a **DPoP proof JWT** with the following structure:
    - Header: `typ: dpop+jwt`, `alg: ES256`, `jwk: {public key}`
    - Payload: `htm` (HTTP method), `htu` (HTTP URI), `iat` (issued at), `jti` (unique identifier)
3. Client sends the DPoP proof as an HTTP header (`DPoP: <proof>`) with the token request.
4. Authorization server binds the issued token to the JWK thumbprint of the client's public key.
5. Server returns `token_type: DPoP` instead of `token_type: Bearer`.
6. For subsequent API calls, the client sends both the `Authorization: DPoP <token>` header and a new `DPoP: <proof>` header specific to that request.

Key Generation

You can generate an ES256 key pair directly in Delphi using the **GenerateDPoPKeyPair** method. This uses OpenSSL internally to create the key pair and automatically sets both `PrivateKey` and `PublicKeyJWK`:

```
OAuth2->DPoPOptions->Enabled = true;
OAuth2->DPoPOptions->Algorithm = dpopES256;
OAuth2->GenerateDPoPKeyPair(); // Generates PrivateKey + PublicKeyJWK automatically
OAuth2->Start();
```

PublicKeyToJWK — Convert Any Key or Certificate to JWK

If you already have a public key, private key, or X.509 certificate (PEM format), you can convert it to JWK using the **PublicKeyToJWK** class function. It accepts any of these inputs and returns the correct JWK JSON:

Input	PEM Header
EC or RSA public key	`-----BEGIN PUBLIC KEY-----`
EC private key	`-----BEGIN EC PRIVATE KEY-----`
RSA private key	`-----BEGIN RSA PRIVATE KEY-----`
X.509 certificate	`-----BEGIN CERTIFICATE-----`

From a PEM key file

```
// From a public key file
OAuth2->DPoPOptions->PublicKeyJWK =
  OAuth2->PublicKeyToJWK(
    TFile::ReadAllText("public_key.pem"));

// From a private key file (extracts the public part)
OAuth2->DPoPOptions->PublicKeyJWK =
  OAuth2->PublicKeyToJWK(
    TFile::ReadAllText("private_key.pem"));
```

From a smart card certificate

Smart cards and HSMs protect the private key — it cannot be read or exported. However, the **certificate** (which contains the public key) is always exportable. Export the certificate as PEM and pass it to `PublicKeyToJWK`:

```
// Export the certificate from the smart card
UnicodeString CertPEM = MySmartCard->ExportCertificatePEM();

// Convert the certificate to JWK
OAuth2->DPoPOptions->PublicKeyJWK =
  OAuth2->PublicKeyToJWK(CertPEM);

// No PrivateKey needed - use OnDPoPSign for signing via the card
OAuth2->DPoPOptions->Enabled = true;
OAuth2->OnDPoPSign = MyDPoPSignHandler;
```

This approach works with any card or HSM that can export its certificate: PKCS#11 tokens, Windows CNG smart cards, USB security keys (YubiKey, Feitian), and cloud HSMs (Azure Key Vault, AWS CloudHSM).

Using OpenSSL command-line tools

You can also generate a key pair externally:

DPoP Options

Property	Description
`DPoPOptions.Enabled`	Set to **True** to enable DPoP proof generation.
`DPoPOptions.Algorithm`	The signing algorithm. Supported values: **ES256** (recommended), **RS256**.
`DPoPOptions.PrivateKey`	The PEM-encoded private key used to sign the DPoP proof JWT.
`DPoPOptions.PublicKeyJWK`	The public key in JWK (JSON Web Key) format, included in the DPoP proof header.

Methods

Method	Description
`GenerateDPoPKeyPair`	Generates an ES256 key pair using OpenSSL and sets both `DPoPOptions.PrivateKey` (PEM) and `DPoPOptions.PublicKeyJWK` (JSON) automatically.
`GetDPoPProof(Method, URL, AccessToken)`	Generates a DPoP proof JWT for the given HTTP method, URL, and access token. Returns the signed JWT string. The proof includes the `ath` claim (access token hash) to bind the proof to the token.

`GetDPoPJWKThumbprint`	Returns the RFC 7638 JWK thumbprint (SHA-256) of the public key, used by the server to bind the token.

Nonce Handling

Some authorization servers require a server-provided nonce in the DPoP proof. When the server responds with a `DPoP-Nonce` header and a 400 or 401 status, the component automatically:

1. Extracts the nonce from the `DPoP-Nonce` response header.
2. Includes the nonce in the `nonce` claim of the next DPoP proof JWT.
3. Retries the request with the updated proof.

Custom Signing with OnDPoPSign (HSM / Smart Card)

In some environments, the private key is stored on a hardware security module (HSM) or smart card (e.g., via PKCS#11 or Windows CNG) and **cannot be extracted**. The `OnDPoPSign` event allows you to perform the signing externally while the component handles everything else (proof structure, nonce, headers).
When `OnDPoPSign` is assigned, the component calls it before attempting internal signing. If you set `Handled := True` and provide the `Signature`, the component uses your signature and skips the internal signing. The `DPoPOptions.PrivateKey` property is not needed in this case.

Event Signature

Parameter	Direction	Description
`SigningInput`	in	The data to sign: `Base64URL(header) + "." + Base64URL(payload)`. This is the standard JWS signing input.
`Algorithm`	in	The algorithm name: `"ES256"` or `"RS256"`.
`Signature`	out	Set this to the Base64URL-encoded signature bytes produced by your signing device.
`Handled`	out	Set to `True` to use your custom signature. If `False`, the component falls back to internal signing using `PrivateKey`.

Smart Card Example

The following example shows how to use DPoP with a signing card. The private key never leaves the card — only the hash is sent to the card for signing.

```
// Configure DPoP - no PrivateKey needed when using OnDPoPSign
OAuth2->DPoPOptions->Enabled = true;
OAuth2->DPoPOptions->Algorithm = dpopES256;
OAuth2->DPoPOptions->PublicKeyJWK =
  "{\"kty\":\"EC\",\"crv\":\"P-256\",\"x\":\"...\",\"y\":\"...\"}";
OAuth2->OnDPoPSign = OnDPoPSignHandler;
OAuth2->Start();

void __fastcall TForm1::OnDPoPSignHandler(TObject *Sender,
  const UnicodeString SigningInput, const UnicodeString Algorithm,
  UnicodeString &Signature, bool &Handled)
{
  // Hash and sign with smart card
  TBytes Hash = GetHashSHA256(SigningInput);
  TBytes Sig = MySmartCardAPI->SignHash(Hash);
  Signature = EncodeBase64URL(Sig);
  Handled = true;
}
```

This approach works with any signing backend: PKCS#11 tokens, Windows CNG (CryptoAPI Next Generation), Azure Key Vault, AWS CloudHSM, or any custom hardware that can produce ECDSA/RSA signatures.

Example

```
// Configure the grant type
OAuth2->OAuth2Options->GrantType = auth2CodePKCE;
OAuth2->OAuth2Options->ClientId = "your-client-id";
OAuth2->AuthorizationServerOptions->AuthURL = "https://provider.com/oauth2/authorize";
OAuth2->AuthorizationServerOptions->TokenURL = "https://provider.com/oauth2/token";
OAuth2->AuthorizationServerOptions->Scope->Text = "openid profile";
// Enable DPoP
OAuth2->DPoPOptions->Enabled = true;
OAuth2->DPoPOptions->Algorithm = "ES256";
OAuth2->DPoPOptions->PrivateKey = LoadPrivateKeyFromFile("dpop_private.pem");
OAuth2->DPoPOptions->PublicKeyJWK = "{\"kty\":\"EC\",\"crv\":\"P-256\",\"x\":\"...\",\"y\":\"...\"}";
OAuth2->Start();
// Generate DPoP proof for API calls
UnicodeString Proof = OAuth2->GetDPoPProof("GET", "https://api.provider.com/resource");
```

OAuth2 | Identity Assertion Authorization Grant

Overview

The Identity Assertion Authorization Grant (IETF draft-ietf-oauth-identity-assertion-authz-grant) performs cross-domain identity and authorization chaining. It lets an application that already holds an access token for a user obtain a new access token for that same user against a different service in another security domain, without any interactive login.
The grant is non-interactive and runs entirely over two automatic back-channel legs. The component handles both legs internally when you call `Start`, so the application only has to configure the endpoints and tokens.

Flow

1. Leg 1, RFC 8693 token exchange. The component POSTs the user's existing access token (`IdentityAssertionOptions.SubjectToken`) and the target `IdentityAssertionOptions.Audience` to the requesting-party authorization server at `IdentityAssertionOptions.RequestingPartyTokenURL`. The server returns an identity assertion JWT, known as the ID-JAG.
2. Leg 2, RFC 7523 jwt-bearer. The component presents that identity assertion to the resource authorization server at `AuthorizationServerOptions.TokenURL`. The server validates the assertion and returns the access token.
3. The access token is surfaced through the normal **OnAfterAccessToken** path, exactly like the other grant types.

If you already hold an identity assertion, set `IdentityAssertionOptions.Assertion` to skip leg 1. The component then presents the supplied assertion directly to the resource authorization server in leg 2.

Identity Assertion Options

Property	Description
`SubjectToken`	The user's existing access token, presented as the subject of the exchange (RFC 8693 `subject_token`).
`SubjectTokenType`	Token type identifier for the subject token. Default `urn:ietf:params:oauth:token-type:access_token`.
`RequestedTokenType`	The token type requested from the requesting-party authorization server. Default `urn:ietf:params:oauth:token-type:id-jag`.
`ActorToken`	Optional. Token representing the acting party when one party acts on behalf of another (RFC 8693 `actor_token`).
`ActorTokenType`	Optional. Token type identifier for the actor token.
`Audience`	The target service for which the identity assertion is requested. Identifies the resource the access token will be used against.
`Resource`	Optional. The resource URI the access token is intended for.
`Scope`	The scope requested for the final access token.

`RequestingPartyTokenURL`	Token endpoint of the requesting-party authorization server where leg 1 (token exchange) is performed.
`RequestingPartyClientId`	Client identifier used to authenticate at the requesting-party authorization server.
`RequestingPartyClientSecret`	Client secret used to authenticate at the requesting-party authorization server.
`Assertion`	Optional. A pre-obtained identity assertion (ID-JAG). When set, leg 1 is skipped and this assertion is presented directly to the resource authorization server.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2IdentityAssertion**.
`AuthorizationServerOptions.TokenURL`	The token endpoint of the resource authorization server, used in leg 2 to redeem the identity assertion for the access token.
`OAuth2Options.ClientId`	The client identifier used to authenticate the leg 2 request at the resource authorization server.
`OAuth2Options.ClientSecret`	The client secret used to authenticate the leg 2 request at the resource authorization server.

Events

Event	Description
`OnBeforeTokenExchange`	Fired before the leg 1 token exchange request is sent. Lets you inspect or modify the URL and parameters, or mark the request as handled.
`OnAfterTokenExchange`	Fired when the requesting-party authorization server returns the identity assertion. Exposes the issued token, its type, and the raw response.
`OnErrorTokenExchange`	Fired when the leg 1 token exchange fails, with the error code, description, and URI.
`OnAfterAccessToken`	Fired when leg 2 completes and the access token is received.
`OnErrorAccessToken`	Fired when a non-recoverable error occurs while obtaining the access token.

Example

Handling the Access Token Event

```
OAuth2->OAuth2Options->GrantType = auth2IdentityAssertion;
OAuth2->IdentityAssertionOptions->RequestingPartyTokenURL = "https://idp.example.com/oauth2/token";
OAuth2->IdentityAssertionOptions->SubjectToken = "the-users-existing-access-token";
OAuth2->IdentityAssertionOptions->Audience = "https://api.partner.com";
OAuth2->IdentityAssertionOptions->Scope = "invoices.read invoices.write";
OAuth2->AuthorizationServerOptions->TokenURL = "https://auth.partner.com/oauth2/token";
OAuth2->OAuth2Options->ClientId = "your-client-id";
OAuth2->OAuth2Options->ClientSecret = "your-client-secret";
OAuth2->Start();
```

When to Use

Use the Identity Assertion Authorization Grant for:

- Cross-domain API access where one application calls a service that lives in another security domain on behalf of the same user.
- Identity and authorization chaining between an enterprise identity provider and a partner resource server.
- Non-interactive token propagation that preserves the original user identity without a fresh login.
- Federated scenarios that follow the ID-JAG model defined by draft-ietf-oauth-identity-assertion-authz-grant.

TsgcHTTP_OAuth2_Server

This component provides the OAuth2 protocol implementation in Server Side Components.

Introduction

This component provides the OAuth2 protocol implementation in Server Side Components.
The server components have a property called Authorization.OAuth.OAuth2 where you can assign an instance of TsgcHTTP_OAuth2_Server, so if Authentication is enabled and OAuh2 property is attached to OAuth2 Server Component, the WebSocket and HTTP Requests require a Bearer Token to be processed, if not the connection will be closed automatically.

```
TsgcHTTP_OAuth2_Server *OAuth2 = new TsgcHTTP_OAuth2_Server(this);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2;
```

The server supports the following authorization types:

- **auth2Code:** It's used to perform authentication and authorization in the majority of application types, including single page applications, web applications, and natively installed applications. The flow enables apps to securely acquire access_tokens that can be used to access resources secured, as well as refresh tokens to get additional access_tokens, and ID tokens for the signed in user.
- **auth2ClientCredentials:** This type of grant is commonly used for server-to-server interactions that must run in the background, without immediate interaction with a user. These types of applications are often referred to as daemons or service accounts.
- **password** (Resource Owner Password Credentials): Allows an application to sign in the user by directly handling their credentials. The client sends the user's username and password to the token endpoint.
- **urn:ietf:params:oauth:grant-type:device_code** (Device Code): Device Authorization Grant per RFC 8628. Enables input-constrained devices (smart TVs, IoT devices) to obtain user authorization by having the user authorize on a secondary device.

The Authorization type can be customized when registering the App, by default, all authorization types are supported.

EndPoints

By default, the component is configured with the following endpoints to handle Authorization and Token request
Authorization: /sgc/oauth2/auth
Token: /sgc/oauth2/token
Revocation: /sgc/oauth2/revoke
Introspection: /sgc/oauth2/introspect
Device Authorization: /sgc/oauth2/device
Device Verification: /sgc/oauth2/device/verify
So if server is listening on port 443 and domain is www.esegece.com, the EndPoints will be:
Authorization: https://www.esegece.com/sgc/oauth2/auth
Token: https://www.esegece.com/sgc/oauth2/token
Revocation: https://www.esegece.com/sgc/oauth2/revoke
Introspection: https://www.esegece.com/sgc/oauth2/introspect
Device Authorization: https://www.esegece.com/sgc/oauth2/device
Device Verification: https://www.esegece.com/sgc/oauth2/device/verify
The endpoints can be configured in OAuth2Options property.
By default, **PKCE** (is an extension to the Authorization Code flow to prevent CSRF and authorization code injection attacks) is enabled.

Configuration

Before you can begin the OAuth2 process, you must register which Apps will be available, this is done using Apps property of OAuth2 server component.

Register App

Use Apps.AddApp to add a new Application to OAuth2 server, you must set the following parameters:

- **App Name:** is the name of the Application. Example: MyApp
- **RedirectURI:** is where the responses will be redirected. Example: http://127.0.0.1:8080
- **ClientId:** is public information and is the ID of the client.
- **ClientSecret:** must be kept confidential.

Optionally you can set the following parameters:

- **ExpiresIn:** by default is 3600 seconds, so the token will expire in 1 hour, you can set a greater value if you need.
- **RefreshToken:** by default refresh tokens are supported, if not, set this parameter to false.
- **AllowedGrantTypes:** by default all grant types are supported (auth2Code and auth2ClientCredentials), but the server can be configured to only allow Code Authorization or only Client Credentials.

Delete App

Use Apps.RemoveApp to delete an existing App.

AddToken

If the server has been restarted while there were some token issued, you can recover these tokens using the method AddToken before starting the OAuth2 Server and after registering the Apps

- **AppName:** the name of the application.
- **Token:** access token.
- **Expires:** when the token expires.
- **RefreshToken:** refresh token.

RemoveToken

Removes an already issued Token.

OAuth2Options

The OAuth2Options property allows configuring the server endpoints and optional features.

Revocation

Token revocation per RFC 7009. When enabled, clients can revoke previously issued access or refresh tokens.

- **OAuth2Options.Revocation.Enabled:** set to True to enable the revocation endpoint.
- **OAuth2Options.Revocation.URL:** the endpoint URL path. Default: /sgc/oauth2/revoke

Introspection

Token introspection per RFC 7662. When enabled, resource servers can query the authorization server to determine the active state and metadata of a token.

- **OAuth2Options.Introspection.Enabled:** set to True to enable the introspection endpoint.
- **OAuth2Options.Introspection.URL:** the endpoint URL path. Default: /sgc/oauth2/introspect

DeviceAuthorization

Device Authorization Grant per RFC 8628. When enabled, input-constrained devices can request authorization by having the user authorize on a secondary device.

- **OAuth2Options.DeviceAuthorization.Enabled:** set to True to enable the device authorization endpoint.
- **OAuth2Options.DeviceAuthorization.URL:** the endpoint URL path for device code requests. Default: /sgc/oauth2/device
- **OAuth2Options.DeviceAuthorization.VerificationURL:** the endpoint URL path for the user verification page. Default: /sgc/oauth2/device/verify

- **OAuth2Options.DeviceAuthorization.ExpiresIn:** the lifetime in seconds of the device code. Default: 600 (10 minutes).
- **OAuth2Options.DeviceAuthorization.Interval:** the minimum polling interval in seconds that the client should use when polling the token endpoint. Default: 5.

Reference

- Properties — 2 properties available on the component.
- Methods — 11 public methods.
- Events — 13 events you can handle.

Guides

- **QuickStart**
  - OAuth2 Server Example
  - OAuth2 Customize Sign-in HTML
  - OAuth2 Server Endpoints
  - OAuth2 Register Apps
  - OAuth2 Recover Access Tokens
- **Authenticate**
  - OAuth2 Server Autentication
  - OAuth2 None Authenticate some URLs

OAuth2 | Server Example

Let's do a simple OAuth2 server example, using a TsgcWebSocketHTTPServer.

First, create a new TsgcWebSocketHTTPServer listening on port 443 and using a self-signed certificate in sgc.pem file.

```
oServer = new TsgcWebSocketHTTPServer(this);
oServer->Port := 80;
oServer->SSLOptions->Port = 443;
oServer->SSLOptions->CertFile = "sgc.pem";
oServer->SSLOptions->KeyFile = "sgc.pem";
oServer->SSLOptions->RootCertFile = "sgc.pem";
oServer->SSL = true;
```

Then create a new instance of TsgcHTTP_OAuth2_Server and assign to previously created server.
Register a new Application with the following values:

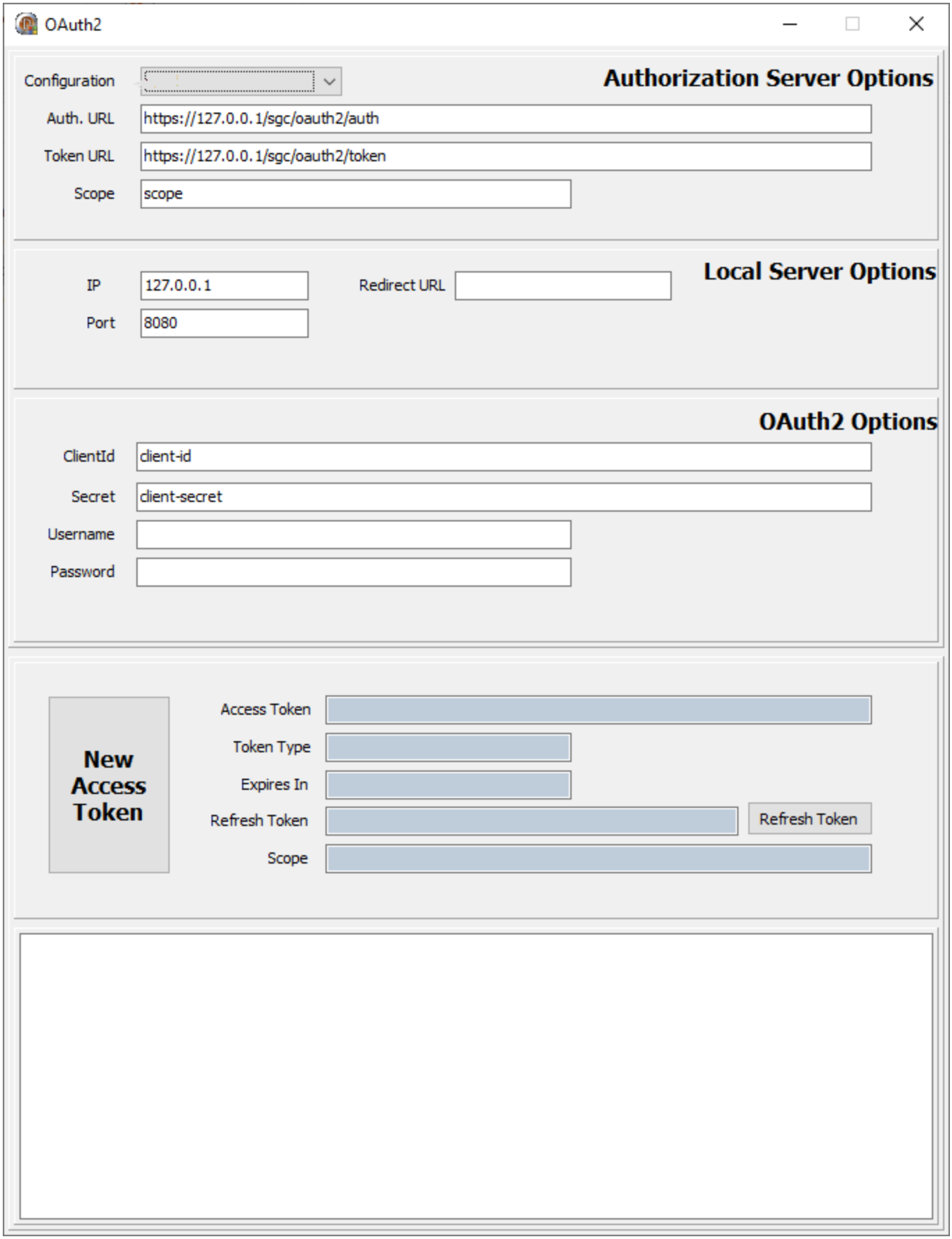
Name: MyApp
RedirectURI: http://127.0.0.1:8080
ClientId: client-id
ClientSecret: client-secret

```
OAuth2 = new TsgcHTTP_OAuth2_Server.Create(this);
OAuth2->Apps->AddApp("MyApp", "http://127.0.0.1:8080", "client-id", "client-secret");
oServer->Authentication->Enabled = true;
oServer->Authentication->OAuth->OAuth2 = OAuth2;
```

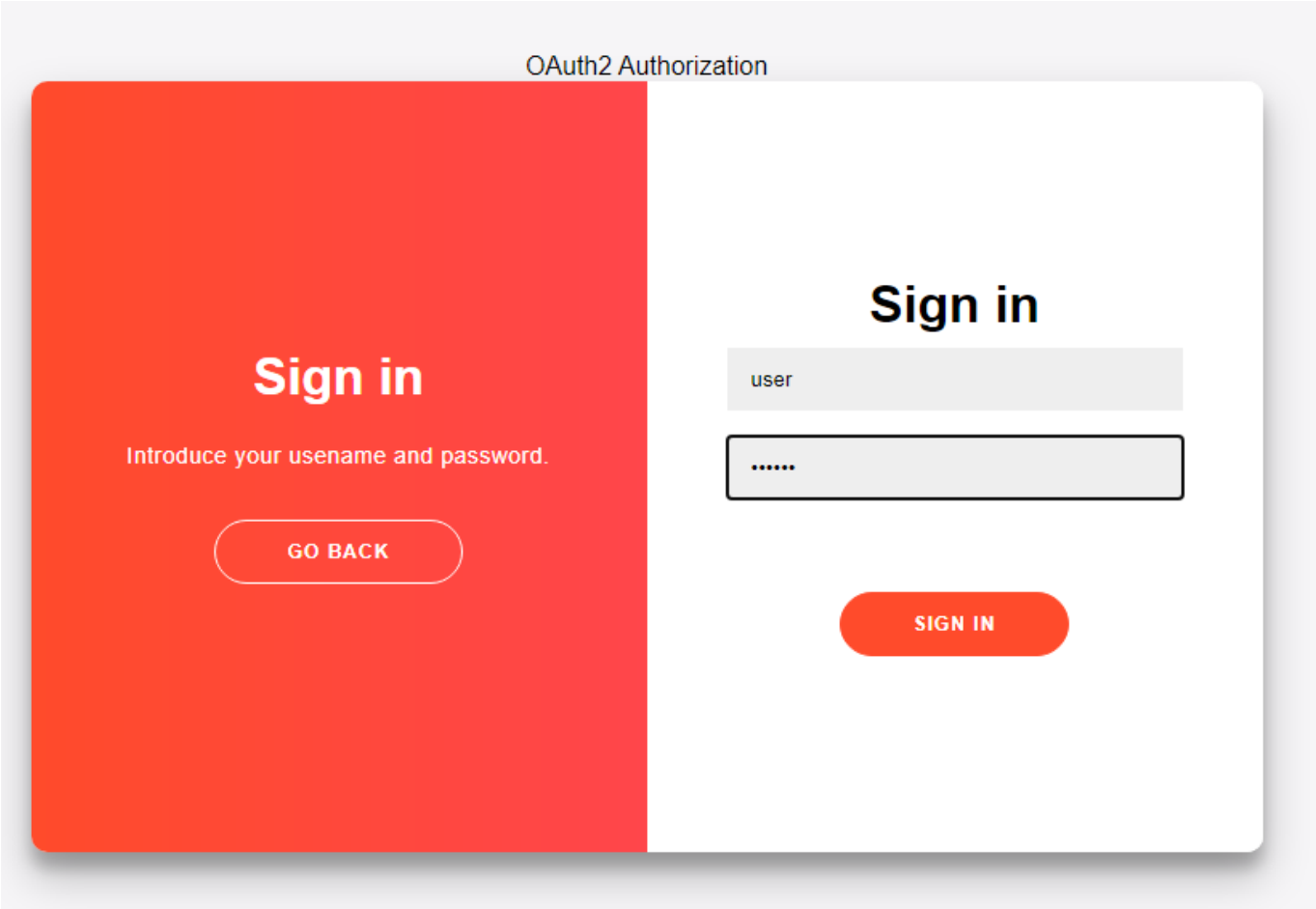
Then handle OnOAuth2Authentication event of OAuth2 server component and implement your own method to login users. I will use the pair "user/secret" to accept a login.

```
void OnOAuth2Authentication(TsgcWSConnection *Connection, TsgcHTTPOAuth2Request *OAuth2, string aUser,
  string aPassword, ref bool Authenticated)
{
  if ((aUser == "user") and (aPassword == "secret"))
  {
    Authenticated = true;
  }
}
```

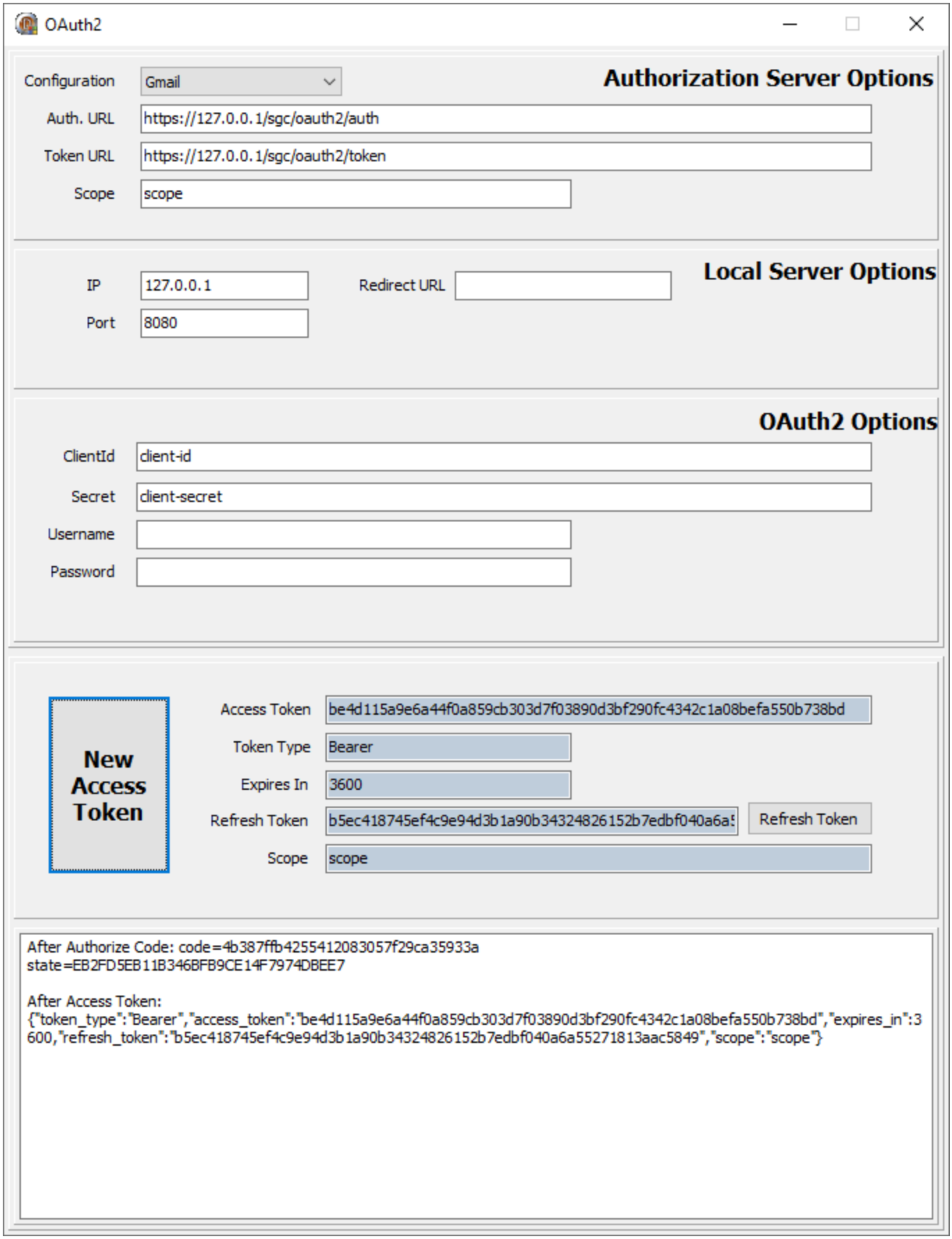
Finally start the server and use a OAuth2 client to login, example you can use the TsgcHTTP_OAuth2_Client included with sgcWebSockets library.



Request a New Access Token, a new Web Browser session will be shown and user must Allow connection and then login.



If login is successful a new Token will be returned to the client. Then all the requests must include this bearer token in the HTTP Headers.


OAuth2
Configuration
Gmail
Authorization Server Options
Auth. URL
https://127.0.0.1/sgc/oauth2/auth
Token URL
https://127.0.0.1/sgc/oauth2/token
Scope
scope
Local Server Options
IP
127.0.0.1
Redirect URL
Port
8080
OAuth2 Options
ClientId
client-id
Secret
client-secret
Username
Password
New Access Token
Access Token
be4d115a9e6a44f0a859cb303d7f03890d3bf290fc4342c1a08befa550b738bd
Token Type
Bearer
Expires In
3600
Refresh Token
b5ec418745ef4c9e94d3b1a90b34324826152b7edbf040a6a5
Refresh Token
Scope
scope
After Authorize Code: code=4b387ffb4255412083057f29ca35933a
state=EB2FD5EB11B346BFB9CE14F7974DBEE7
After Access Token:
{"token_type":"Bearer","access_token":"be4d115a9e6a44f0a859cb303d7f03890d3bf290fc4342c1a08befa550b738bd","expires_in":3600,"refresh_token":"b5ec418745ef4c9e94d3b1a90b34324826152b7edbf040a6a55271813aac5849","scope":"scope"}

OAuth2 | Customize Sign-In HTML

When an OAuth2 client do a request to get a new Access Token, a Web-Page is shown in a web-browser to Allow this connection and login with an User and Password.
The HTML page is included by default in Server component, but this code can be customized using **OnAuth2BeforeDispatchPage** event.

```
void OnOAuth2BeforeDispatchPage(TObject *Sender; TsgcHTTPOAuth2Request *OAuth2; ref string HTML)
{
  HTML = "your custom html";
}
```

If you customize your HTML with a completely new HTML code, at least you must maintain the form where the Username and password are sent:

```
<form action="">
<input type="hidden" name="request_type" value="sign-in" />
<input type="username" name="username" placeholder="Username" />
<input type="password" name="password" placeholder="Password" />
<input type="hidden" name="id" value="" />
<p></p>
<button>Sign In</button>
</form>
```

The id parameter, which is hidden, must maintain the same value of the original form to allow server identify the request.

OAuth2 | Server Endpoints

By default, the OAuth2 Server uses the following Endpoints:
Authorization: /sgc/oauth2/auth
Token: /sgc/oauth2/token
Revocation: /sgc/oauth2/revoke (POST) - Revokes tokens per RFC 7009
Introspection: /sgc/oauth2/introspect (POST) - Returns token metadata per RFC 7662
Device Authorization: /sgc/oauth2/device (POST) - Issues device codes per RFC 8628
Device Verification: /sgc/oauth2/device/verify (GET/POST) - User verification page
Which means that if your server listens on IP 80.54.41.30 and port 8443, the full OAuth2 Endpoints will be:
Authorization: https://80.54.41.30:8443/sgc/oauth2/auth
Token: https://80.54.41.30:8443/sgc/oauth2/token
Revocation: https://80.54.41.30:8443/sgc/oauth2/revoke
Introspection: https://80.54.41.30:8443/sgc/oauth2/introspect
Device Authorization: https://80.54.41.30:8443/sgc/oauth2/device
Device Verification: https://80.54.41.30:8443/sgc/oauth2/device/verify
This Endpoints can be modified easily, just access to OAuth2Options property of component and modify Authorization and Token URLs.
Example: if your endpoints must be
Authorization: https://80.54.41.30:8443/authentication/auth
Token: https://80.54.41.30:8443/authentication/token
Set the OAuth2Options property with the following values:
OAuth2Options.Authorization.URL = /authentication/auth
OAuth2Options.Token.URL = /authentication/token
The same approach applies to the other endpoints:
OAuth2Options.Revocation.URL = /authentication/revoke
OAuth2Options.Introspection.URL = /authentication/introspect
OAuth2Options.DeviceAuthorization.URL = /authentication/device
OAuth2Options.DeviceAuthorization.VerificationURL = /authentication/device/verify

OAuth2 | Register Apps

Before a new OAuth2 is requested by a client, the App must be registered in the server.
Register a new App requires the following information:

- **App Name:** is the name of the Application. Example: MyApp
- **RedirectURI:** is where the responses will be redirected. Example: http://127.0.0.1:8080
- **ClientId:** is public information and is the ID of the client.
- **ClientSecret:** must be kept confidential.

Optionally you can set the following parameters:

- **ExpiresIn:** by default is 3600 seconds, so the token will expire in 1 hour, you can set a greater value if you need.
- **RefreshToken:** by default refresh tokens are supported, if not, set this parameter to false.

If a new client wants to authenticate using OAuth2, first the App requires to be registered in the server, you can use:

1. RegisterApp
This method requires the App Name and RedirectURI, and will return a ClientId and ClientSecret.

2. Apps.AddApp
This method requires AppName, RedirectURI, ClientId and ClientSecret. Usually you can use this method when a server has some already created Apps and you want to load them before is started.

Both methods do the same, register the Application in the server, but first is most useful when the App is registered the first time and second method when you want to load all registered apps before start the server (because are saved on database for example).

OAuth2 | Recover Access Tokens

If the OAuth2 Server is destroyed (because it's restarted) and there are valid Access Tokens issued, these are lost by default. You can recover these Access Tokens using the method **AddToken**. This method stores the tokens in the OAuth2 Server.

Add a Token requires the following information:

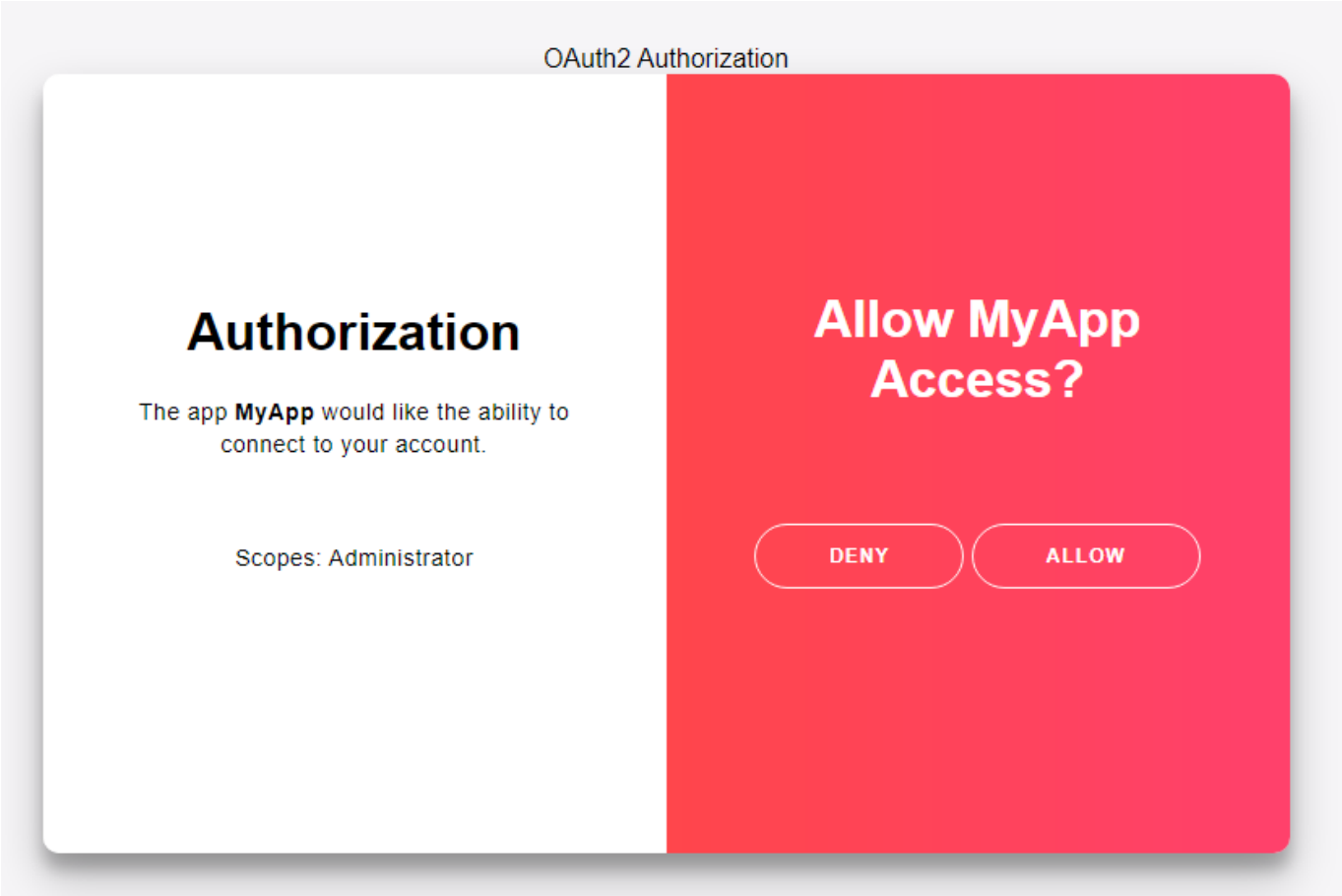
- **AppName:** the name of the app.
- **Token:** access token.
- **Expires:** when the token expires.
- **RefreshToken:** refresh token.

You can save the issued tokens handling the **OAuth2AfterAccessToken** event.

```
private void OnOAuth2AfterAccessToken(TObject *Sender, TsgcWSConnection *Connection, TsgcHTTPOAuth2Request *OAuth
  string aResponse)
{
  // ... store OAuth2 Token data
}

OAuth2 = new TsgcHTTP_OAuth2_Server.Create(this);
OAuth2->Apps->AddApp("MyApp", "http://127.0.0.1:8080", "client-id", "client-secret");
OAuth2->AddToken("MyApp", "12146ce12b0e4813987f2794f768905cefc39da6fbd54f6d9b38387489280608", EncodeDate(2022,1,1
  "ef3e3dfa56ec44109c3d345b1541f08e539ce21432d9433099b48a3d08d34bc0");
oServer->Authentication->Enabled = true;
oServer->Authentication->OAuth->OAuth2 = OAuth2;
```

OAuth2 | Server Authentication

When an OAuth2 client requests a new Authorization, the server shows a web page where the user must allow the connection and then login. This page is provided by sgcWebSockets library and is dispatched automatically when a client requests an Authorization.



If the user Allows the access, a login form will be shown where the user must set the Username and Password. This data will be received OnOAuth2Authentication event, so you must validate than the user/password is correct and if it is, then set Authenticated parameter to true.

```
void OnOAuth2Authentication(TsgcWSConnection *Connection, TsgcHTTPOAuth2Request *OAuth2, string aUser,
  string aPassword, ref bool Authenticated)
{
  if ((aUser == "user") and (aPassword == "secret"))
  {
    Authenticated = true;
  }
}
```

OAuth2 | None Authenticate URLs

By default, when OAuth2 is enabled on Server Side, all the HTTP Requests require Authentication using Bearer Tokens.
If you want allow some URLs to be accessed without the need of use a Bearer Token, you can use the event **OnOAuth2BeforeRequest**

```
procedure OnOAuth2BeforeRequest(TObject *Sender; TsgcWSConnection *aConnection; TStringList *aHeaders;
  ref bool Cancel)
{
  if (DecodeGETFullPath(aHeaders) == "/Public.html")
  {
    Cancel = true;
  }
}
```

OAuth2 Server | Authorization Code Grant

Overview

The Authorization Code grant is the most common OAuth2 flow for server-side applications. The TsgcHTTP_OAuth2_Server provides two endpoints to handle this flow: an Authorization endpoint that displays a sign-in page to the user, and a Token endpoint that exchanges the authorization code for an access token.
This flow is suitable for web applications, desktop applications, and mobile applications where the client can interact with a browser.

Flow

1. Client redirects the user to `/sgc/oauth2/auth?response_type=code&client_id=...&redirect_uri=...&scope=...&state=...`
2. Server fires **OnOAuth2BeforeDispatchPage** to allow customization of the HTML sign-in page.
3. User submits credentials. Server fires **OnOAuth2Authentication** to validate the username and password.
4. If authenticated, the server redirects the user to the `redirect_uri` with `?code=...&state=...` appended.
5. Client POSTs to `/sgc/oauth2/token` with `grant_type=authorization_code&code=...&redirect_uri=...&client_id=...&client_secret=...`
6. Server validates the authorization code and client credentials, fires **OnOAuth2AfterAccessToken**, and returns a JSON response containing the `access_token`, `token_type`, `expires_in`, and optionally a `refresh_token`.

Configuration

Property	Description
`OAuth2Options.Authorization.Enabled`	Set to True to enable the authorization endpoint. Default: True.
`OAuth2Options.Authorization.URL`	The authorization endpoint URL path. Default: /sgc/oauth2/auth
`OAuth2Options.Token.Enabled`	Set to True to enable the token endpoint. Default: True.
`OAuth2Options.Token.URL`	The token endpoint URL path. Default: /sgc/oauth2/token
`OAuth2Options.PKCE`	When True, the server requires PKCE (code_challenge and code_verifier) for authorization code requests. Default: True.

App Registration

Before clients can use the Authorization Code flow, you must register the application on the server. Use `Apps.AddApp` and ensure the `AllowedGrantTypes` includes `auth2Code`.

```
OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  TsgcOAuth2GrantTypes() << auth2Code);
```

Events

Event	Description
`OnOAuth2BeforeDispatchPage`	Fired before the sign-in HTML page is sent to the user. Allows customization of the HTML content.
`OnOAuth2Authentication`	Fired when the user submits credentials. Validate the username and password here and set `Authenticated` to True or False.
`OnOAuth2AfterAccessToken`	Fired after the server successfully issues an access token. Useful for logging.
`OnOAuth2AfterValidateAccessToken`	Fired when a client makes a request with a valid, non-expired token. Useful for logging and auditing.

Example

The following example shows a complete server setup with the Authorization Code grant, including app registration and event handlers.

```
// Create and configure OAuth2 server
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);

// Register an application
OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  TsgcOAuth2GrantTypes() << auth2Code);

// Attach to HTTP server
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;

void __fastcall TForm1::OAuth2ServerOAuth2Authentication(TObject *Sender,
  const UnicodeString User, const UnicodeString Password, bool &Authenticated)
{
  Authenticated = (User == "admin") && (Password == "secret");
}

void __fastcall TForm1::OAuth2ServerOAuth2AfterAccessToken(TObject *Sender,
  const UnicodeString Token, const UnicodeString RefreshToken, int ExpiresIn)
{
  Log("Access token issued: " + Token);
}
```

PKCE Support

When `OAuth2Options.PKCE` is set to True (the default), the server enforces Proof Key for Code Exchange. The client must include a `code_challenge` parameter in the authorization request and a corresponding `code_verifier` in the token request. The server validates that the code_verifier matches the original code_challenge before issuing a token.
PKCE is recommended for all clients, especially public clients (native apps and SPAs) that cannot securely store a client secret.

OAuth2 Server | Client Credentials Grant

Overview

The Client Credentials grant is used for machine-to-machine (M2M) communication where no user interaction is required. The client authenticates directly with the TsgcHTTP_OAuth2_Server using its own credentials (client_id and client_secret) and receives an access token. There is no authorization code step, no redirect, and no user login page.
This grant type is appropriate when the client application is acting on its own behalf rather than on behalf of a specific user.

Flow

1. Client POSTs to the token endpoint with `grant_type=client_credentials&client_id=...&client_secret=...&scope=...`
2. Server validates the client credentials against the registered application.
3. Server fires **OnOAuth2AfterAccessToken** and returns a JSON response with `access_token`, `token_type`, and `expires_in`.

Note that no `OnOAuth2Authentication` event is fired for this grant type, since there is no user to authenticate. The server validates the client_id and client_secret against the registered app automatically.

Configuration

Register the application with `AllowedGrantTypes` that includes `auth2ClientCredentials`. Only the token endpoint is needed for this flow.

Property	Description
`OAuth2Options.Token.Enabled`	Set to True to enable the token endpoint. Default: True.
`OAuth2Options.Token.URL`	The token endpoint URL path. Default: /sgc/oauth2/token

App Registration

```
OAuth2Server->Apps->AddApp("MyServiceApp", "",
  "service-client-id", "service-client-secret", 3600, false,
  TsgcOAuth2GrantTypes() << auth2ClientCredentials);
```

The `RedirectURI` can be left empty since no redirect is involved. `RefreshToken` is typically set to False because the client can request a new token at any time using its credentials.

Events

Event	Description
`OnOAuth2AfterAccessToken`	Fired after the server successfully issues an access token. Useful for logging.
`OnOAuth2AfterValidateAccessToken`	Fired when a client makes a request with a valid token. Useful for auditing.

Example

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->Apps->AddApp("BackendService", "",
  "service-client-id", "service-client-secret", 7200, false,
  TsgcOAuth2GrantTypes() << auth2ClientCredentials);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;
void __fastcall TForm1::OAuth2ServerOAuth2AfterAccessToken(TObject *Sender,
  const UnicodeString Token, const UnicodeString RefreshToken, int ExpiresIn)
{
  Log("Service token issued: " + Token);
}
```

When to Use

Use the Client Credentials grant on the server for:

- Backend services and daemons that authenticate without user interaction.
- Server-to-server API communication.
- Microservice-to-microservice authentication.
- Automated processes, scheduled jobs, and batch operations.

OAuth2 Server | Resource Owner Password Credentials Grant

Overview

The Resource Owner Password Credentials (ROPC) grant allows the client to send the user's username and password directly to the token endpoint. The TsgcHTTP_OAuth2_Server validates the credentials via the **OnOAuth2Authentication** event and issues an access token if authentication succeeds.
This flow does not involve a browser redirect or a sign-in page. The client collects credentials directly from the user and submits them to the server.

Security Warning

This grant type is deprecated in OAuth 2.1. It should only be used when other flows (such as Authorization Code with PKCE) are not feasible, for example in legacy applications or highly trusted first-party clients. The Authorization Code grant with PKCE is the recommended alternative.

Flow

1. Client POSTs to the token endpoint with `grant_type=password&username=...&password=...&client_id=...&client_secret=...&scope=...`
2. Server fires **OnOAuth2Authentication** to validate the username and password.
3. If authenticated, the server fires **OnOAuth2AfterAccessToken** and returns a JSON response with `access_token`, `token_type`, `expires_in`, and optionally a `refresh_token`.
4. If authentication fails, the server returns an error response.

Configuration

Register the application with `AllowedGrantTypes` that includes `auth2ResourceOwnerPassword`.

Property	Description
`OAuth2Options.Token.Enabled`	Set to True to enable the token endpoint. Default: True.
`OAuth2Options.Token.URL`	The token endpoint URL path. Default: /sgc/oauth2/token

App Registration

```
OAuth2Server->Apps->AddApp("MyApp", "",
  "my-client-id", "my-client-secret", 3600, true,
  TsgcOAuth2GrantTypes() << auth2ResourceOwnerPassword);
```

Events

Event	Description

`OnOAuth2Authentication`	Fired when the server receives the username and password. Validate credentials here and set `Authenticated` to True or False.
`OnOAuth2AfterAccessToken`	Fired after the server successfully issues an access token. Useful for logging.
`OnOAuth2AfterValidateAccessToken`	Fired when a client makes a request with a valid token.

Example

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->Apps->AddApp("MyApp", "",
  "my-client-id", "my-client-secret", 3600, true,
  TsgcOAuth2GrantTypes() << auth2ResourceOwnerPassword);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;
void __fastcall TForm1::OAuth2ServerOAuth2Authentication(TObject *Sender,
  const UnicodeString User, const UnicodeString Password, bool &Authenticated)
{
  Authenticated = ValidateUser(User, Password);
}
void __fastcall TForm1::OAuth2ServerOAuth2AfterAccessToken(TObject *Sender,
  const UnicodeString Token, const UnicodeString RefreshToken, int ExpiresIn)
{
  Log("Token issued for user: " + Token);
}
```

When to Use

Use the Resource Owner Password Credentials grant only when:

- The client is a highly trusted first-party application.
- Other grant types (Authorization Code with PKCE) are not feasible.
- Migrating legacy applications that already collect user credentials directly.

For new applications, always prefer the Authorization Code grant with PKCE.

OAuth2 Server | Refresh Token Grant

Overview

The Refresh Token grant allows a client to exchange a previously issued refresh token for a new access token without requiring user interaction. The TsgcHTTP_OAuth2_Server validates the refresh token and issues a new access token (and optionally a new refresh token).
This is useful for long-lived sessions where the access token has a short expiration but the client needs continued access to protected resources.

Flow

1. Client POSTs to the token endpoint with `grant_type=refresh_token&refresh_token=...&client_id=...&client_secret=...&scope=...`
2. Server validates the refresh token against its stored tokens.
3. Server fires **OnOAuth2AfterRefreshToken** and returns a JSON response with a new `access_token`, `token_type`, `expires_in`, and optionally a new `refresh_token`.

Configuration

Refresh tokens are enabled per application when registering the app. Set the `RefreshToken` parameter to True in `Apps.AddApp`.

Property	Description
`OAuth2Options.Token.Enabled`	Set to True to enable the token endpoint. Default: True.
`OAuth2Options.Token.URL`	The token endpoint URL path. Default: /sgc/oauth2/token
`RefreshToken (AddApp parameter)`	Set to True to issue refresh tokens for this application. Default: True.

App Registration

When registering the app, set the `RefreshToken` parameter to True so that the server issues refresh tokens alongside access tokens.

```
// RefreshToken = true (6th parameter)
OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  TsgcOAuth2GrantTypes() << auth2Code);
```

Events

Event	Description
`OnOAuth2AfterRefreshToken`	Fired after the server successfully issues a new access token from a refresh token. Useful for logging token rotation.
`OnOAuth2AfterAccessToken`	Fired after the new access token is issued.

Example

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  TsgcOAuth2GrantTypes() << auth2Code);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;
void __fastcall TForm1::OAuth2ServerOAuth2AfterRefreshToken(TObject *Sender,
  const UnicodeString OldToken, const UnicodeString NewToken,
  const UnicodeString NewRefreshToken, int ExpiresIn)
{
  Log("Token refreshed. New token: " + NewToken);
}
```

Token Rotation

When a refresh token is used, the server may issue a new refresh token alongside the new access token. The old refresh token is invalidated. This is known as refresh token rotation and helps prevent token replay attacks.

Recovering Tokens After Restart

If the server is restarted while tokens are still valid, you can recover them using `Apps.AddToken` before starting the server. See OAuth2 Recover Access Tokens for details.

OAuth2 Server | Device Authorization Grant (RFC 8628)

Overview

The Device Authorization Grant is designed for input-constrained devices (smart TVs, IoT devices, CLI tools) that cannot easily display a browser or accept keyboard input. The TsgcHTTP_OAuth2_Server provides a device authorization endpoint and a verification page so that users can authorize the device from a secondary device such as a phone or computer.

Flow

1. Device POSTs to `/sgc/oauth2/device` with `client_id=...&scope=...`
2. Server fires **OnOAuth2DeviceAuthorization** and returns a JSON response with `device_code`, `user_code`, `verification_uri`, `expires_in`, and `interval`.
3. Device displays the user_code and verification_uri to the user (e.g., "Go to https://example.com/sgc/oauth2/device/verify and enter code: ABCD-1234").
4. User navigates to the verification URI on a secondary device and enters the user_code.
5. Server fires **OnOAuth2DeviceCodeVerification** to validate the user_code and authorize the device.
6. Meanwhile, the device polls the token endpoint with `grant_type=urn:ietf:params:oauth:grant-type:device_code&device_code=...&client_id=...`
7. Server returns `authorization_pending` until the user authorizes. Once authorized, it returns the access token.

Configuration

Property	Description
`OAuth2Options.DeviceAuthorization.Enabled`	Set to True to enable the device authorization endpoint. Default: False.
`OAuth2Options.DeviceAuthorization.URL`	The device authorization endpoint URL path. Default: /sgc/oauth2/device
`OAuth2Options.DeviceAuthorization.VerificationURL`	The verification page URL path where the user enters the code. Default: /sgc/oauth2/device/verify
`OAuth2Options.DeviceAuthorization.ExpiresIn`	The lifetime in seconds of the device code. Default: 600 (10 minutes).
`OAuth2Options.DeviceAuthorization.Interval`	The minimum polling interval in seconds that the device should use. Default: 5.

Events

Event	Description
`OnOAuth2DeviceAuthorization`	Fired when a device requests a device code. Allows customizing the response or logging the request.

`OnOAuth2DeviceCodeVerification`	Fired when a user submits a verification code on the device verification page. Validate the user_code and authorize or deny the device.
`OnOAuth2AfterAccessToken`	Fired after the server issues the access token to the device.

Example

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->OAuth2Options->DeviceAuthorization->Enabled = true;
OAuth2Server->OAuth2Options->DeviceAuthorization->ExpiresIn = 600;
OAuth2Server->OAuth2Options->DeviceAuthorization->Interval = 5;
OAuth2Server->Apps->AddApp("DeviceApp", "",
  "device-client-id", "device-client-secret", 3600, true,
  TsgcOAuth2GrantTypes() << auth2Code);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;
void __fastcall TForm1::OAuth2ServerOAuth2DeviceCodeVerification(TObject *Sender,
  const UnicodeString UserCode, bool &Authorized)
{
  Authorized = true;
}
void __fastcall TForm1::OAuth2ServerOAuth2AfterAccessToken(TObject *Sender,
  const UnicodeString Token, const UnicodeString RefreshToken, int ExpiresIn)
{
  Log("Device authorized. Token: " + Token);
}
```

Polling Behavior

While the user has not yet authorized the device, the device polls the token endpoint at the configured interval. The server responds with:

- `authorization_pending` -- the user has not yet completed authorization.
- `slow_down` -- the device is polling too frequently. The device should increase the interval by 5 seconds.
- `expired_token` -- the device code has expired. The device must restart the flow.
- `access_denied` -- the user denied authorization.

Once the user authorizes the device, the next poll returns the access token.

OAuth2 Server | Token Revocation (RFC 7009)

Overview

Token revocation allows clients to notify the TsgcHTTP_OAuth2_Server that a previously issued access token or refresh token is no longer needed. The server invalidates the token so it can no longer be used to access protected resources.
This is useful when a user logs out, when an application is uninstalled, or when a token may have been compromised.

Endpoint

POST to `/sgc/oauth2/revoke` with the following parameters:

Parameter	Description
`token`	The token to revoke (required).
`token_type_hint`	Optional hint about the token type: `access_token` or `refresh_token`.

Behavior

Per RFC 7009, the server always returns HTTP 200 OK regardless of whether the token was found or successfully revoked. This prevents token existence leakage -- a client cannot determine whether a token was valid by observing the response status.
When a refresh token is revoked, the associated access token may also be invalidated (implementation-dependent). When an access token is revoked, the associated refresh token remains valid unless explicitly revoked.

Configuration

Property	Description
`OAuth2Options.Revocation.Enabled`	Set to True to enable the revocation endpoint. Default: False.
`OAuth2Options.Revocation.URL`	The revocation endpoint URL path. Default: /sgc/oauth2/revoke

Events

Event	Description
`OnOAuth2AfterRevokeToken`	Fired after a token revocation attempt. Provides the token value, token_type_hint, and a `Revoked` parameter indicating whether the token was successfully invalidated. Useful for logging revocation activity.

Example

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);

OAuth2Server->OAuth2Options->Revocation->Enabled = true;

OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  TsgcOAuth2GrantTypes() << auth2Code);

Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;

void __fastcall TForm1::OAuth2ServerOAuth2AfterRevokeToken(TObject *Sender,
  const UnicodeString Token, const UnicodeString TokenTypeHint, bool &Revoked)
{
  Log("Token revoked: " + Token + " (type: " + TokenTypeHint + ")");
  Revoked = true;
}
```

Client Request Example

A client revokes a token by sending a POST request to the revocation endpoint:

```
// Client-side: revoke an access token
HTTPClient->Post("https://example.com/sgc/oauth2/revoke",
  "token=eyJhbGciOi...&token_type_hint=access_token");
```

OAuth2 Server | Token Introspection (RFC 7662)

Overview

Token introspection allows resource servers to query the TsgcHTTP_OAuth2_Server to determine the active state and metadata of an access token or refresh token. This is useful when a resource server needs to validate a token without having to decode it locally.

Endpoint

POST to `/sgc/oauth2/introspect` with the following parameters:

Parameter	Description
`token`	The token to introspect (required).
`token_type_hint`	Optional hint about the token type: `access_token` or `refresh_token`.

Response

The introspection endpoint returns a JSON object with the following fields:

Field	Description
`active`	Boolean. True if the token is currently active (valid and not expired or revoked).
`scope`	The scope associated with the token.
`client_id`	The client identifier for the application that requested the token.
`token_type`	The type of the token (e.g., Bearer).
`exp`	The expiration time of the token (Unix timestamp).
`username`	The username associated with the token (if applicable).

If the token is invalid, expired, or revoked, the response contains only `{"active": false}`.

Configuration

Property	Description
`OAuth2Options.Introspection.Enabled`	Set to True to enable the introspection endpoint. Default: False.

`OAuth2Options.Introspection.URL`	The introspection endpoint URL path. Default: /sgc/oauth2/introspect

Events

Event	Description
`OnOAuth2AfterIntrospectToken`	Fired after a token introspection request. Provides the token value and a `IsActive` parameter that can be modified to override the default validation result. Useful for custom validation logic and auditing.

Example

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);

OAuth2Server->OAuth2Options->Introspection->Enabled = true;

OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  TsgcOAuth2GrantTypes() << auth2Code);

Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;

void __fastcall TForm1::OAuth2ServerOAuth2AfterIntrospectToken(TObject *Sender,
  const UnicodeString Token, bool &IsActive)
{
  Log("Token introspected: " + Token + " Active: " + BoolToStr(IsActive));
}
```

Resource Server Request Example

A resource server validates a token by sending a POST request to the introspection endpoint:

```
// Resource server: introspect a token
UnicodeString Response = HTTPClient->Post("https://example.com/sgc/oauth2/introspect",
  "token=eyJhbGciOi...&token_type_hint=access_token");
// Response: {"active":true,"scope":"read write","client_id":"my-client-id",...}
```

OAuth2 Server | DPoP Validation (RFC 9449)

Overview

DPoP (Demonstrating Proof of Possession) per RFC 9449 enables the TsgcHTTP_OAuth2_Server to require sender-constrained tokens. When DPoP is enabled, the server validates DPoP proof JWTs, binds tokens to the client's JWK thumbprint, and returns `token_type: DPoP` instead of `Bearer`.
This ensures that access tokens can only be used by the client that originally obtained them, preventing token theft and replay attacks.

How It Works

1. Client includes a `DPoP` header with a signed JWT proof in token requests.
2. Server validates the proof JWT:
   - `typ` must be `dpop+jwt`
   - Algorithm (`alg`) must be asymmetric (e.g., ES256, RS256)
   - `htm` (HTTP method) and `htu` (HTTP URI) must match the request
   - `iat` (issued at) must be recent (freshness check)
   - Signature is verified against the JWK embedded in the proof header
3. Server extracts the JWK thumbprint and binds it to the issued token.
4. Token response includes `token_type: DPoP` instead of `Bearer`.
5. On subsequent resource requests, the server validates the DPoP proof including the `ath` claim (access token hash) to ensure the proof is bound to the correct token.

Configuration

Property	Description
`OAuth2Options.DPoP`	Boolean. Set to True to enable DPoP-bound token validation. Default: False.

Signature Verification

The server reconstructs the EC or RSA public key from the JWK included in the DPoP proof header and verifies the JWT signature. Only asymmetric algorithms are accepted (ES256, ES384, ES512, RS256, RS384, RS512, PS256, PS384, PS512). Symmetric algorithms (HS256, etc.) are rejected.
By default the signature is verified with OpenSSL. When `sgcCryptoBackend` selects the native backend (or sgcWebSockets is built with `SGC_NATIVE_CRYPTO`), it is verified with the native sgcCrypto routines and no OpenSSL library is needed, for ES256, ES384, ES512, RS256, RS384 and RS512. On that path the JWK curve must match the algorithm (P-256 for ES256, P-384 for ES384, P-521 for ES512) and the EC public point must lie on the curve. See Running without OpenSSL.

Nonce Support

The server can issue a `DPoP-Nonce` header in token responses. When present, the client must include the nonce value in the `nonce` claim of subsequent DPoP proof JWTs. This provides an additional layer of replay protection.

Events

Event	Description

`OnOAuth2ValidateDPoP`	Fired after standard DPoP validation completes. Allows custom validation logic such as checking additional claims or enforcing policy. Set `Valid` to False to reject the proof.

Example

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->OAuth2Options->DPoP = true;
OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  TsgcOAuth2GrantTypes() << auth2Code);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;
void __fastcall TForm1::OAuth2ServerOAuth2ValidateDPoP(TObject *Sender,
  const UnicodeString DPoPProof, const UnicodeString AccessToken,
  const UnicodeString Method, const UnicodeString URL, bool &Valid)
{
  Log("DPoP proof validated for: " + Method + " " + URL);
  Valid = true;
}
```

Invalid Proof Rejection

When DPoP proof validation fails, the server returns HTTP 400 with an error response:
`{"error": "invalid_dpop_proof", "error_description": "..."}`

Common reasons for rejection include:

- Missing or malformed DPoP header.
- Invalid JWT signature.
- Symmetric algorithm used (only asymmetric algorithms are allowed).
- Mismatched `htm` or `htu` claims.
- Expired or future `iat` claim.
- Mismatched `ath` (access token hash) on resource requests.
- Invalid or missing `nonce` when the server requires it.

TsgcHTTP_OAuth2_Server_Provider

This component allows you to integrate External OAuth2 Providers (like Azure AD, Google, Facebook.

Introduction

This component allows you to integrate External OAuth2 Providers (like Azure AD, Google, Facebook...) in your server component (like an HTP server), so an user can login using the Azure AD credentials and if the authentication is successful, the HTTP server can provide access to protected resources.

The server components have a property called Authorization.OAuth.OAuth2Provider where you can assign an instance of TsgcHTTP_OAuth2_Server_Provider, so if Authentication is enabled and OAuh2Provider property is attached to OAuth2 Provider Server Component, the WebSocket and HTTP Requests require a Cookie / Bearer Token to be processed, if not the connection will be closed automatically.

```
TsgcHTTP_OAuth2_Server_Provider *OAuth2Provider = new TsgcHTTP_OAuth2_Server_Provider(this);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2Provider = OAuth2Provider;
```

Register OAuth2 Provider

Before the server is started, you must configure the OAuth2 Providers that the server will use to authenticate. Use the method **RegisterProvider** to configure the OAuth2 Providers, this method has the following parameters:

- **Name:** is the name of the provider, it can be any name, is just to identify the provider later.
- **ClientId:** is the public client Id, this value is provided by the OAuth2 Provider.
- **ClientSecret:** is the private client secret (must be keep confidential), this value is provided by the OAuth2 Provider.
- **AuthorizeURL:** is the URL where the OAuth2 client will redirect to login (the connection is using a web browser).
- **TokenURL:** is the URL the server will use to validate the token provided after a successful authorization (the connection is server to server).
- **Scope:** is the value of the scope/s.
- **URL:** is the URL of the HTTP Server that will be used to redirect to the Authorization URL.
- **CallbackURL:** is the URL configured in the OAuth2 Provider that will process the response sent from the OAuth2 server after a successful Authorization.

Example: to configure Azure AD, it requires a tenant-id which is added to the OAuth2 URLs, ClientId, ClientSecret, Scope and a CallbackURL.

```
RegisterProvider(
  'azure',
  '90945b8d-f6b7-4b97-b2bd-21c3c90b5f3x',
  'PN67Q~5m06c-~X_GMyMf9zMntmm5l2dt~3jVq',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/authorize',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/token',
  'user.read',
  '/login',
  'https://localhost/callback'
  );
```

To delete an existing Provider, use the method **UnRegisterProvider**.

Reference

- Properties — 2 properties available on the component.
- Methods — 15 public methods.
- Events — 4 events you can handle.

Guides

- **QuickStart**
  - OAuth2 Provider Azure AD
- **Authenticate**
  - OAuth2 Provider Private Endpoints
  - OAuth2 Provider Authentication
  - OAuth2 Provider Requests

OAuth2 Provider | Azure AD

Azure AD uses the following OAuth2 Authorization URLs

Authorization: https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/authorize
Token: https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token

The **<tenant-id>** must be replaced by your own values.

When you create the OAuth2 configuration, you must configure a server callback url, this url will be used by Azure to send a response to your server after a successful authorization.

Example: find below a simple example of how register Azure AD provider.

Values provided by Azure AD

ClientId: 90945b8d-f6b7-4b97-b2bd-21c3c90b5f3x
ClientSecret: PN67Q~5m06c-~X_GMyMf9zMntmm5l2dt~3jVq
tenant: a0ca2055-5dd1-467f-bf13-291f6fd715c6
scope: user.read
CallbackURL: https://localhost/callback

How Register Azure AD

```
RegisterProvider(
  'azure',
  '90945b8d-f6b7-4b97-b2bd-21c3c90b5f3x',
  'PN67Q~5m06c-~X_GMyMf9zMntmm5l2dt~3jVq',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/authorize',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/token',
  'user.read',
  '/login',
  'https://localhost/callback'
  );
```

OAuth2 Provider | Private Endpoints

Every time the Server receives a HTTP Request, the event **OnOAuth2IsPrivateEndpoint** is called to ask if the Endpoint is private or not. By default, is not private.

```
void OnOAuth2IsPrivateEndpoint(TObject *Sender, const string aEndpoint, ref bool IsPrivate)
{
  if (aEndpoint == "/private")
  {
    IsPrivate = True;
  }
}
```

OAuth2 Provider | Authentication

The OAuth2 Provider Server Component allows you to authenticate using an external OAuth2 provider (like Azure AD or Google) to access the protected resources of your server. **Example:** you can configure your HTTP server to allow users to log in using Azure credentials; if the login is successful, those users will be allowed to access the protected resources of your server.

The Authentication process is done from the server side and the OAuth2 tokens are not shared with the clients, this means that when the user logins using Azure for example, if the authentication is successful, Azure returns an Access Token that allows you to send requests to the Azure server to get some information (depending of the scope) about the user profile, emails... This Access Token **IS NOT SHARED** with the client (example a web-browser), instead of returning the Access token to the client, the server creates a random ID that it's linked internally with the Access Token, so every time the Client (Web Browser) wants to do a call to the OAuth2 Server, uses the public ID and the server uses this ID to get the OAuth2 Access Token to proxy the HTTP Requests.

Find below an example of how the OAuth2 Authentication works. The example will use the Azure AD configuration described in the following link OAuth2 Provider Azure AD.

Start the Server

The server starts listening on localhost and port 443. The sgcWebSockets HTTP Server is linked to the OAuth2 Server Provider Component and the Authentication property is enabled.
Before the server is started, the Azure OAuth2 Provider is registered using the following method call.

```
RegisterProvider(
  'azure',
  '90945b8d-f6b7-4b97-b2bd-21c3c90b5f3x',
  'PN67Q~5m06c-~X_GMyMf9zMntmm5l2dt~3jVq',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/authorize',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/token',
  'user.read',
  '/login',
  'https://localhost/callback'
  );
```

User Logins

The user opens a new web browser and go to '/login' endpoint.
The server detects that the '/login' endpoint is used to login using the Azure provider so redirects to

https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/authorize

And the OAuth2 authentication Flow Starts.

OAuth2 Authentication

The user is redirected to the OAuth2 Server Authentication Endpoint, now he must login using the credentials and accept the terms of the OAuth2 Application.

If the authorization is successful, Azure AD sends a Code to the url

https://localhost/callback

Validate the OAuth2 Code

Now, the server has received a code from Azure and it will do an internal connection to Azure (from server to server) to validate this token is correct (and avoid someone is trying to hack the server).

The server connects to

https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/token

Passing some paramenters like the code received and the clientsecret, if the validation is successful, Azure returns the Access Token that can be used to access the Azure Protected Resources like read the profile, email...

Successful Access Token

When the server receives a success full AccessToken, the event **OnOAuth2ProviderTokenValid** is called, so here you can configure how the AccessToken is stored (if it is) accessing to the parameter class TsgcHTTPOAuth2ProviderToken

AccesToken: is the OAuth2 Token returned by Azure
ID: is the public identifier stored as a cookie.

In this event you can configure what to do after a successful authentication, example: if you want to redirect the user to the private url, use the following

Response.Redirect.URL := 'https://localhost/private';

Send Requests to Azure

Now, you can send requests to the Azure server using the Public ID stored as a cookie.
Example: if you want to read the profile data, use the following method.

Get('ID', 'https://graph.microsoft.com/v1.0/me');

Where ID is the public ID identifier.

OAuth2 Provider | Requests

Once the Authentication has been successful, you can send requests to the OAuth2 Protected Server using the Public ID Token stored as a cookie.

The OAuth2 Provider Server Component, has several methods to send HTTP Requests: GET, POST, DELETE...

You can pass the Token as a parameter or pass the RequestInfo class if you are using the Indy Server components.

```
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Document == "/private"
  {
    // return OAuth2 profile data
    AResponseInfo->ContentText = OAuth2Provider->Get(ARequestInfo, "https://graph.microsoft.com/v1.0/me");
    AResponseInfo->ContentType = "application/json";
    AResponseInfo->ResponseNo = 200;
  }
  else
  {
    AResponseInfo->ResponseNo = 404;
  }
}
```

HTTP | JWT

JWT (JSON Web Token) typically consists of a header + payload + signature.

Header

Contains the metadata information about the JWT.

- **alg:** the algorithm used to sign the token.
- **typ:** the type of the token, always JWT.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

You can find more headers, but the previous ones will always be present.

Payload

The payload contains the claims of the JWT. The standard headers are the following:

- **iss:** the issuer, the entity that generates and issues the JWT.
- **sub:** the subject, the entity identified by this token.
- **aud:** the audience, the target audience for this JWT.
- **exp:** the expiry, the timestamp in UNIX format after which the token should not be accepted.
- **iat:** issued at, specifies the date when the token was issued.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Signature

The signature is created using the Encoded Header, Encoded Payload, a Secret and a Cryptographic Algorithm.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0
NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5M
```

```
DIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Algorithms supported

The following algorithms are supported by both Client and Server JWT components.

- HS256
- HS384
- HS512
- RS256
- RS384
- RS512
- ES256
- ES384
- ES512
- ML-DSA-44

- ML-DSA-65
- ML-DSA-87

OpenSSL libraries are required to sign and verify the JWT, with one exception.

ML-DSA, post-quantum signatures

The three ML-DSA algorithms are the post-quantum JOSE algorithms of RFC 9964, and they are the exception above: they are implemented in pure Object Pascal in the sgcCrypto pack, so they sign and validate with no OpenSSL library present. The signature is FIPS 204 ML-DSA in the pure mode with an empty context over the JWS signing input, carried in the token as base64url. The signer is deterministic, so the same key and the same input always produce the same signature.

The enum values are `jwtMLDSA44`, `jwtMLDSA65` and `jwtMLDSA87`, and the `alg` header they write is `ML-DSA-44`, `ML-DSA-65` and `ML-DSA-87`. The client signs with `JWTOptions.Algorithms.MLDSA.PrivateKey`, a PKCS#8 PEM private key, and the server validates with `JWTOptions.Algorithms.MLDSA.PublicKey`, a SubjectPublicKeyInfo PEM public key. The parameter set of the key has to be the one the algorithm names: signing raises otherwise, and validating returns False.

Signatures are large. An ML-DSA-44 signature is 2420 bytes against 64 for ES256, so a token grows by roughly 3.2 KB of base64url, and an ML-DSA-87 one by about 6.2 KB. Check the header size limits of anything the token passes through before adopting it.

Keys can also be carried as AKP JSON Web Keys, RFC 9964 sections 3 and 4. `sgcMLDSA_ExportPublicJWK` and `sgcMLDSA_ExportPrivateJWK` write one, `sgcMLDSA_ImportJWK` reads either form, and `sgcMLDSA_ImportJWKAsPEM` converts a JWK straight into the PEM strings the two components read. A private JWK stores the 32-byte seed in `priv`, the only private form section 4 defines, so an expanded private key cannot be written as a JWK. All four functions are declared in `sgcHTTP_JWT_MLDSA`.

* The ML-DSA algorithms need the sgcCrypto pack. Without it the enum values still exist, so a form stored with one still loads, but signing raises and every ML-DSA token is rejected.

Components

- **TsgcHTTP_JWT_Client:** JWT client which allows you to encode and sign JWT and send as an Authorization Header in **HTTP** and **WebSocket** protocols.
- **TsgcHTTP_JWT_Server:** JWT server which allows you to decode and validate JWT received as an Authorization Header in **HTTP** and **WebSocket** protocols.

** JWT Components require at least Indy version 10.6.0.5169 or sgcWebSockets Enterprise Edition.*

TsgcHTTP_JWT_Client

Configure which openSSL libraries you will use when using JWT client.

Introduction

The TsgcHTTP_JWT_Client component allows you to encode and sign JWT Tokens, attached to a WebSocket Client or HTTP/2 client, the token will be sent automatically as an Authorization Bearer Token Header.

Configuration

You can configure the JWT values in the **JWTOptions** properties, there are 2 main properties: **Header** and **Payload**, just set the values for every required property.

If the Signature is encrypted using a Private Key (RS and ES algorithms), set the value in the **PrivateKey** property of the Algorithm.
If the Signature is encrypted using a Secret (HS algorithms), set the value in the **Secret** property of the Algorithm.

OpenSSL Options

Configure which openSSL libraries you will use when using JWT client.

OpenSSL_Options: configuration of the openSSL libraries.
- **APIVersion:** allows defining which OpenSSL API will be used.
  - **oslAPI_1_0:** uses API 1.0 OpenSSL, it's latest supported by Indy
  - **oslAPI_1_1:** uses API 1.1 OpenSSL, requires our custom Indy library and allows you to use OpenSSL 1.1.1 libraries (with TLS 1.3 support).
  - **oslAPI_3_0:** uses API 3.0 OpenSSL, requires our custom Indy library and allows you to use OpenSSL 3.0.0 libraries (with TLS 1.3 support).
- **LibPath:** here you can configure where are located the openSSL libraries
  - **oslpNone:** this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.
  - **oslpDefaultFolder:** sets automatically the openSSL path where the libraries should be located for all IDE personalities.
  - **oslpCustomFolder:** if this is the option selected, define the full path in the property LibPathCustom.
- **LibPathCustom:** when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.
- **UnixSymLinks:** enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
  - **oslsSymLinksDefault:** by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).
  - **oslsSymLinksLoadFirst:** Load SymLinks and do before trying to load the version libraries.
  - **oslsSymLinksLoad:** Load SymLinks after trying to load the version libraries.
  - **oslsSymLinksDontLoad:** don't load the SymLinks.

Custom Headers

The Header and Payload properties contains the most common headers used to generate a JWT, but you can add more headers calling the method **AddKeyValue** and passing the Key and Value as parameters.
Example: if you want add a new record in the JWT Header with your name, use the following method

```
Header->AddKeyValue("name", "John Smith");
```

After configuring the properties, to generate the JWT, just call the method **Sign** and will return the value of the JWT.

WebSocket Client and JWT

TsgcWebSocketClient allows the use of JWT when connecting to WebSocket servers, just create a new JWT client and assign to **Authentication.Token.JWT** property.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient->URL = "ws://www.esegece.com:2052";


TsgcHTTP_JWT_Client oJWT = new TsgcHTTP_JWT_Client();
oJWT->JWTOptions->Header.alg = jwtRS256;
oJWT->JWTOptions->Payload.sub = "1234567890";
oJWT->JWTOptions->Payload.iat = 1516239022;


oClient->Authentication->Enabled = true;
oClient->Authentication->URL->Enabled = false;
oClient->Authentication->Token->Enabled = true;
oClient->Authentication->Token->JWT = oJWT;
oClient->Active = true;
```

HTTP Clients and JWT

TsgcHTTP2Client and TsgcHTTP1Client allows the use of JWT when connecting to HTTP/2 servers, just create a new JWT client and assign to **Authentication.Token.JWT** property.

```
TsgcHTTP2Client oHTTP = new TsgcHTTP2Client();


TsgcHTTP_JWT_Client oJWT = new TsgcHTTP_JWT_Client();
oJWT->JWTOptions->Header->alg = jwtRS256;
oJWT->JWTOptions->Payload->sub = "1234567890";
oJWT->JWTOptions->Payload->iat = 1516239022;


oHTTP->Authentication->Token->JWT = oHTTP;
oHTTP->Get("https://your.api.com");
```

Expiration

The Authorization Token can be **re-created every time** you send an HTTP request using an HTTP client or can be **reused several times till it expires**.
Example: calling Apple APNs using Tokens, requires that the token is reused at least during 20 minutes and at a maximum of 1 hour. Use the Property **RefreshTokenAfter** to set the seconds when the token will expire, for example after 30 minutes.

```
RefreshTokenAfter = 60 * 40.
```

Create JWT Signature

You can **create JWT Signatures manually** to use on applications that doesn't make use of WebSocket or HTTP Protocol, or if you are using components from third-parties applications and you only need the JWT Token.

In order to obtain the JWT Signature, just create a new instance of the JWT Client and fill the properties manually, when all properties are set, call the method **Sign** and it will return the JWT Token.

```
TsgcHTTP_JWT_Client oJWT = new TsgcHTTP_JWT_Client(this);
// ... header
oJWT->JWTOptions->Header->alg = jwtHS256;
oJWT->JWTOptions->Algorithms->HS->Secret = "79F66F1E-E998-436B-8A0A-3E5DEFA4FD9E";
// ... payload
oJWT->JWTOptions->Payload->jti = "9B66FB94-B761-42B1-A1AF-3C44233DBE87";
oJWT->JWTOptions->Payload->iat = 1630925658;
oJWT->JWTOptions->Payload->iss = "2886EC7547B7BA6A9009";
oJWT->JWTOptions->Payload->exp = 1630933158;
// ... custom payload values
oJWT->JWTOptions->Payload->ClearKeyValues;
oJWT->JWTOptions->Payload->AddKeyValue("origin", "www->yourwebsite->com");
oJWT->JWTOptions->Payload->AddKeyValue("ip", "69.39.46.178");
// ... get JWT Token
ShowMessage(oJWT->Sign());
```

Reference

- Properties — 2 properties available on the component.
- Methods — 2 public methods.
- Events — 0 events you can handle.

TsgcHTTP_JWT_Server

TsgcHTTP_JWT_Server component.

Introduction

The TsgcHTTP_JWT_Server component allows you to **decode** and **validate** JWT tokens received in **WebSocket Handshake** when using WebSocket protocol or as **HTTP Header** when using HTTP protocol.

Configuration

You can configure the following properties in the **JWTOptions** property of the component:

If the Signature is validated using a Public Key (RS and ES algorithms), set the value in the **PublicKey** property of the Algorithm.
If the Signature is validated using a Secret (HS algorithms), set the value in the **Secret** property of the Algorithm.

To validate JWT tokens, just **attach a TsgcHTTP_JWT_Server** instance to **Authentication.JWT.JWT** property of the WebSocket/HTTP Server.

```
TsgcWebSocketHTTPServer oServer = new TsgcWebSocketHTTPServer();
oServer->Port = 80;
TsgcHTTP_JWT_Server oJWT = new TsgcHTTP_JWT_Server();
oJWT->JWTOptions->Algorithms->RS->PublicKey->Text = "public key here";
oServer->Authorization->Enabled = true;
oServer->Authorization->JWT->JWT = oJWT;
oServer->Active = true;
```

Claim Validations

Once the signature has been verified, the server validates the registered claims of the Payload. The **exp**, **nbf** and **iat** claims are checked, earlier versions only verified the signature, so an expired token was accepted. A claim which is not present in the Payload is not asserted by the issuer and is not validated.

The checks are configured in the **JWTOptions.Validations** property:

Expiration: rejects the token when the **exp** claim is in the past. Enabled by default, set **Validations.Expiration := False** to disable the expiry check.
NotBefore: rejects the token when the **nbf** claim is in the future. Enabled by default.
Issued: rejects the token when the **iat** claim is in the future. Enabled by default.
Leeway: seconds of clock skew allowed when the exp, nbf and iat claims are compared with the current time. The default is 60 seconds, set it to zero to compare the claims with no allowance.

Reference

- Properties — 2 properties available on the component.
- Methods — 3 public methods.
- Events — 7 events you can handle.

WebAuthn

WebAuthn (Web Authentication) is a web standard developed by the World Wide Web Consortium (W3C) and FIDO Alliance to enable **secure**, **passwordless authentication on the web**. It is part of the broader FIDO2 framework and aims to **reduce reliance on traditional passwords**, which are often vulnerable to phishing, credential stuffing, and other attacks.

At its core, **WebAuthn allows users to authenticate using public-key cryptography. Instead of a username and password, users register a unique public-private key pair with a web application** (the Relying Party). The private key is securely stored on an authenticator—such as a hardware security key, smartphone, or built-in biometric device—while the public key is stored on the server.

During authentication, the server issues a challenge that must be signed by the user's private key. The signed challenge is returned and verified using the stored public key, ensuring both the integrity and origin of the response. This approach prevents credentials from being intercepted or reused.

WebAuthn supports a range of authenticators and devices, making it flexible for both developers and users. It also enables multi-factor authentication (MFA) when combined with other factors like PINs or biometrics, significantly improving security without sacrificing usability.

Server

- **TsgcWSAPIServer_WebAuthn:** The component provides a simple but powerful solution to implement the WebAuthn Relying Party server, enabling passwordless authentication in your web application. The server has passed the full **FIDO Conformance Test** using the Conformance Self-Validation Testing tool.

Client

- **Javascript Client:** act as the bridge between the user, browser, and authenticator during registration and authentication

Passkeys

Passkeys replace passwords with a key pair created by the authenticator of the user: Windows Hello, iCloud Keychain, Google Password Manager, a password manager or a FIDO2 security key. The private key never leaves the authenticator, the server stores only the public key, and the signature is bound to the domain of the site, so passkeys can not be phished, reused or leaked from the server database.
Technically a passkey is a **discoverable WebAuthn credential** (a resident key): the authenticator keeps the user handle next to the private key, so the user can sign in without typing a user name, and the browser can offer the passkeys of the site in the autofill list of the user name field. The TsgcWSAPIServer_WebAuthn server supports passkeys with the settings and events described below.

Availability

The WebAuthn server is included in **sgcWebSockets Enterprise** and in the **sgcAuth** pack. WebAuthn needs a secure context: serve the pages over https (or `localhost`) and set **WebAuthnOptions.RelyingParty** to the host name shown in the browser.

Register discoverable credentials

The property **WebAuthnOptions.DefaultOptions.Registration.DiscoverableCredential** sets the `residentKey` requested in the registration options:

Value	residentKey	Meaning
`waundcPreferred` (default)	`preferred`	The authenticator creates a discoverable credential when it can.
`waundcRequired`	`required`	Only discoverable credentials are accepted. Use it for passkeys and usernameless sign in.
`waundcDiscouraged`	`discouraged`	The authenticator should create a server-side credential, which needs the user name to sign in.

A registration request can override the default with the field `discoverable_credential` (`"required"`, `"preferred"` or `"discouraged"`). The setting is only applied when the request does not ask for a specific `residentKey` itself.

Several passkeys per user

A user usually has several passkeys, for example one on the computer and one on the phone. Register each of them with the same user name: when the user name already has credentials known by the server, the registration reuses their user handle (`user.id`), so every passkey of the account shares one user handle. The server knows the credentials registered while it runs and the ones added with AddCredential, so when your credentials live in a database, add them at startup to keep one user handle per account. With a user name, the authentication options contain every passkey of the user in `allowCredentials`.

Usernameless sign in

Request the authentication options with an empty user name. The options then carry no `allowCredentials`, the browser shows the passkeys it has for the relying party and the user picks one. The server requires the `userHandle` of the response and checks that it belongs to the credential used, and a credential which is not known by the application is rejected.

```
// ... "Sign in with a passkey" button: the browser lists the passkeys of the site
const options = await post("/sgcWebAuthn/Authentication/Options", {username: ""});
const assertion = await SimpleWebAuthnBrowser.startAuthentication(options);
const result = await post("/sgcWebAuthn/Authentication/Verify", assertion);
```

Passkey autofill

With conditional mediation the passkeys appear in the autofill list of the user name field, next to the saved passwords. Add `autocomplete="username webauthn"` to the input, load `/sgcWebAuthn.js` (served by the component, endpoint **EndpointsOptions.WebAuthn**) and call `startAuthentication(options, true)` with usernameless options when the page loads. The promise resolves when the user picks a passkey.

```
<script src="/sgcWebAuthn.js"></script>
<input id="username" name="username" autocomplete="username webauthn">
<script>
async function post(url, body) {
  const r = await fetch(url, {method: "POST",
    headers: {"Content-Type": "application/json"}, body: JSON.stringify(body)});
  return r.json();
}
async function passkeyAutofill() {
  if (!(await SimpleWebAuthnBrowser.browserSupportsWebAuthnAutofill())) return;
  // ... no user name: the options carry no allowCredentials
  const options = await post("/sgcWebAuthn/Authentication/Options", {username: ""});
  // ... true: conditional mediation, the passkeys appear in the autofill list
  const assertion = await SimpleWebAuthnBrowser.startAuthentication(options, true);
  const result = await post("/sgcWebAuthn/Authentication/Verify", assertion);
  if (result.verified) { location.href = "/"; }
}
passkeyAutofill();
</script>
```

Autofill needs a browser that supports conditional mediation (current Chrome, Edge and Safari). `browserSupportsWebAuthnAutofill()` returns False in the other browsers, keep a "Sign in with a passkey" button for them.

Store the passkeys in your own database

The component does not persist credentials. Four events connect it to your storage:

Event	What to do
OnWebAuthnRegistrationSuccessful	Save `aCredentialRecord`, for example `aCredentialRecord.AsJSON`, together with its **CredentialId** and **Username**.
OnWebAuthnAuthenticationOptionsRequest	With a user name, add the passkeys of that user to `CredentialRecords`. Without a user name, add nothing.
OnWebAuthnAuthenticationGetCredential	Fires when the credential used by the browser is not in the credential list of the ceremony, which is the case of a usernameless or autofill sign in. Look the credential id up, fill `aCredentialRecord` and set `Found` to True.
OnWebAuthnAuthenticationSuccessful	Save the updated **SignCount** and **BackupState** of `aAuthentication.Credential.CredentialRecord` and create the session.

The record returned by **OnWebAuthnAuthenticationGetCredential** must have the requested **CredentialId**, and the same **Username** when the ceremony was started with a user name, otherwise the sign in fails. Keep the **UserId** (user handle) in the stored record, it is compared with the `userHandle` sent by the authenticator. The events run in the server connection threads, protect shared resources accordingly.

Synced or device-bound

The authenticator data flags tell what kind of passkey the server received. They are stored in the credential record as **BackupEligible** (BE flag) and **BackupState** (BS flag):

BackupEligible	BackupState	Meaning

True	True	Synced passkey, backed up by the platform or the password manager and available on the other devices of the user.
True	False	Multi-device passkey that is not backed up yet.
False	False	Device-bound passkey, for example a security key or a TPM key that never leaves the device. Recommend the user to register a second passkey.

At every sign in the server rejects a response with the BS flag set and the BE flag not set, and a response whose BE flag differs from the stored **BackupEligible**, because the eligibility of a credential never changes. The new BS flag is copied to **BackupState**, save it in **OnWebAuthnAuthenticationSuccessful**.

Signature counter and cloned authenticators

Some authenticators increase a signature counter at every sign in. When the counter of the response or the stored **SignCount** is not zero, the counter of the response must be greater than the stored value, otherwise the sign in is rejected because the authenticator may have been cloned. The stored value only moves forward, and it must be saved after every sign in for the detection to work. Synced passkeys usually always report 0, which disables the check for them. Set **WebAuthnOptions.AllowSignCountLessOrEqualStoredValue** to True to accept a counter that does not increase (the stored value is still never lowered).

Demo

The demo **Demos\26.Authentication\01.Passkeys** registers several passkeys per user, signs in without a user name and with autofill, shows whether every passkey is synced or device-bound, and stores the credentials in a JSON file of the application through the events above. See also WebAuthn and the JavaScript client.

TsgcWSAPIServer_WebAuthn

Only the OpenSSL 3.

Introduction

The **TsgcWSAPIServer_WebAuthn** component provides a simple but powerful solution to implement the WebAuthn Relying Party server, enabling passwordless authentication in your web application. A WebAuthn application consists of a WebAuthn server that handles the server-side registration and authentication and a client-side application that usually is a javascript application.

WebAuthn requires the use of secure connections (SSL/TLS), so the OpenSSL libraries must be deployed and configured with the server.

Only the OpenSSL 3.0.0+ API is supported, so previous OpenSSL versions may not work.

Configuration

The **TsgcWSAPIServer_WebAuthn** must be attached to an HTTP server, TsgcWebSocketHTTP_Server or TsgcWebSocketServer_HTTPAPI using the Server property. You can configure the server endpoints that will handle the registration and authentication options, and the WebAuthn options like supported algorithms, origins, and more.

Endpoints Options

Here you can configure the server endpoints that will handle the HTTP/JavaScript requests to use WebAuthn as an authenticator. The component is already configured with default endpoints, but you can change all of them to fit your needs.

- **AuthenticationOptions:** by default is /sgcWebAuthn/Authentication/Options
- **AuthenticationVerify:** by default is /sgcWebAuthn/Authentication/Verify
- **RegistrationOptions:** by default is /sgcWebAuthn/Registration/Options
- **RegistrationVerify:** by default is /sgcWebAuthn/Registration/Verify
- **Webauthn:** includes the javascript library used by default. You can disable this property and use your own webauthn library.
- **Test:** by default is disabled, only use to test the WebAuthn functionality.

Example: if your server is listening on domain www.test.com, the request to authentication options by default will be http://www.test.com/sgcWebAuthn/Authentication/Options

WebAuthn Options

In this property you can configure the main options of the WebAuthn Server Component.

- **RelyingParty:** a **mandatory** property where the **DNS name of the server** must be defined. Example: if the server is running on the domain www.test.com, set this property to "www.test.com".

WebAuthn uses origins to enforce same-origin policy constraints, which are essential for preventing phishing and cross-site attacks. During the WebAuthn registration and authentication processes, the origin is strictly validated by the browser and the authenticator.

- **Origins:** If the requests can come from different origins, use the property Origin to set the additional origins. Example: if the requests can come from login.test.com and www.test.co.uk, configure the Origins property with the values: https://login.test.com and https://www.test.co.uk

- **TopOrigins:** Normally, WebAuthn relies on the origin of the calling frame (i.e., the one invoking navigator.credentials.create() or navigator.credentials.get()). However, web pages can be embedded in iframes, which might come from a different origin than the top-level page. This opens up potential for abuse or clickjacking-style attacks. To mitigate this, the WebAuthn Level 2 spec introduces TopOrigin where you can define the TopOrigins.

In WebAuthn, crossOrigin is a boolean parameter that indicates whether the WebAuthn operation is being performed from a cross-origin context, such as an iframe embedded from a different origin than the top-level browsing context.
This parameter was introduced to help browsers and authenticators safely handle authentication requests in embedded environments—a common scenario in modern web applications.

- **AllowCrossOrigins:** if true, indicates that requests made from a cross-origin iframe (e.g., an iframe at https://auth.example.com embedded in a page at https://app.example.org) are allowed. By default, it is disabled.

WebAuthn supports a variety of **cryptographic algorithms** for public key credential generation and verification. These algorithms are used during credential registration (with navigator.credentials.create()) and authentication (with navigator.credentials.get()), and they ensure secure signing and validation of challenges using asymmetric key pairs. The server is configured by default with the **ES256 and RS256** which are the most common algorithms. You can change at any time which algorithms are supported from the **Algorithms** property. The following algorithms are supported:

- ES256
- ES384
- ES512
- RS256
- RS384
- RS512
- PS256
- PS384
- PS512
- RS1
- EdDSA

In WebAuthn, **attestation is an optional mechanism** that allows the authenticator (e.g., device or security key) to provide information about its manufacturer, model, and security characteristics during credential creation. This information helps the Relying Party (RP) decide whether to trust the authenticator.

Different attestation formats define how this data is structured and verified. Three commonly used formats are android-key, packed, and others like fido-u2f, apple, or none. By default, all attestation formats are enabled. You can find below the list of supported attestation formats:

- **NoneAttestation:** in this case none attestation data is returned. Prioritizes user privacy by avoiding the exposure of device identifiers. Common in applications that don't care about device provenance.

- **PackedAttestation:** is a flexible, compact format used by many authenticators. The authenticator returns an attestation certificate and signature. Can be: **Full attestation:** Signed with a vendor-provided key and cert or **Self attestation:** Signed using the credential private key. Most widely used across different platforms (e.g., YubiKey, Windows Hello).

- **TPMAttestation:** Used by devices with a Trusted Platform Module (TPM). Attestation is signed using keys from the TPM and includes a certificate chain. Used by Enterprise desktops/laptops with TPM chips (e.g., Windows machines).

- **AndroidKeyAttestation:** Used by Android devices with the Android Keystore. The key is generated in hardware, and attestation includes information signed by a certificate chain issued by the device manufacturer. Used by Android phones with hardware-backed keystores (TEE or StrongBox).

- **AppleAttestation:** Used by Apple platform authenticators, such as Touch ID and Face ID. Attestation is generated by Apple's internal APIs and includes a special certificate format. Used on Safari using Apple biometrics.

- **FidoU2FAttestation:** Legacy attestation format used by FIDO U2F authenticators. Returns a U2F-compatible certificate and signature. Used by older security keys (e.g., early YubiKeys) that support FIDO U2F.

In the WebAuthn API, **AllowCredentials** is an **optional field** used during the authentication process (via navigator.credentials.get()). It **specifies a list of credential IDs that are permitted to authenticate the user** for a particular Relying Party (RP). This mechanism lets the RP control which credentials are considered valid for a login attempt. The property credentials has the following fields:

- **AllowCredentials:** if enabled (false by default) specifies a list of credential IDs that are permitted to authenticate the user

- **ExcludeCredentials:** given a username, shows all existing credentials already stored in the server component.

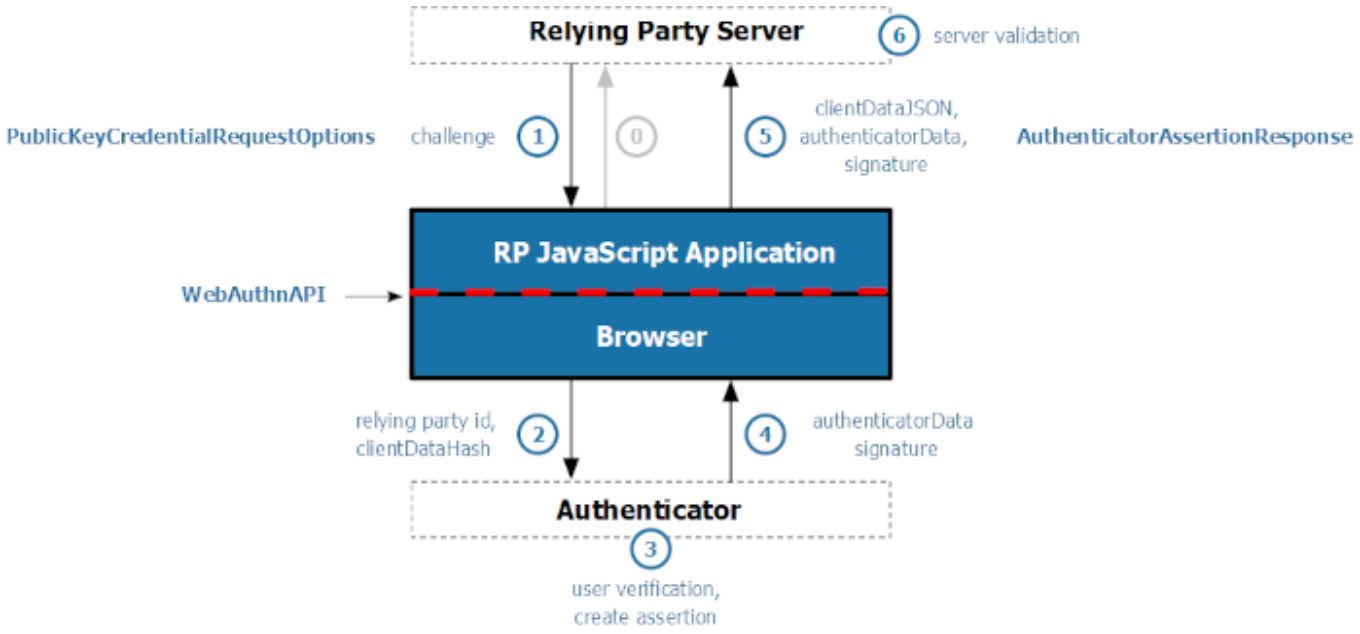
- **Limit:** the max number of credentials that will be sent when ExcludeCredentials is true.

WebAuthn Protocol

- **WebAuthn Registration:** The server generates a challenge and sends it to the client, which uses an authenticator (e.g. security key or biometric device) to create a key pair. The public key is sent back and stored by the server for future authentication. Find below more info about the registration flow events:
  - **WebAuthn Registration Request**
  - **WebAuthn Registration Response**
  - **WebAuthn Registration Result**

- **WebAuthn Authentication:** The server sends a challenge to the client, which signs it using the previously registered private key stored in the authenticator. The signed response is verified by the server using the stored public key to confirm the user's identity. Find below more info about the authentication flow events:
  - **WebAuthn Authentication Request**
  - **WebAuthn Authentication Response**
  - **WebAuthn Authentication Result**

- **MDS:** The FIDO Alliance Metadata Service (MDS) is a centralized repository of Metadata Statements that is used by relying parties to validate authenticator attestation and prove the genuineness of the device model.



- **Authorization:** The client can request a bearer token from the server during the authentication flow. This token can be used later to open a new WebSocket or HTTP connection without the need to log in using passkeys.
    - **WebAuthn Authorization WebSocket**
    - **WebAuthn Authorization HTTP**

Reference

- Properties — 4 properties available on the component.
- Methods — 8 public methods.
- Events — 16 events you can handle.

WebAuthn | Registration

WebAuthn Registration involves the **client** (browser), the **authenticator** (security device), and the **relying party** (RP; TsgcWSAPIServer_WebAuthn delphi/cbuilder server).

Registration Options

Creating a new user credential usually requires the client starting the registration flow using a new HTTP Request to the **Registration Options Endpoint** configured in the TsgcWSAPIServer_WebAuthn server. By default, the end-point is /sgcWebAuthn/Registration/Options, so if your server is listening in the domain https://www.test.com, the client should make a new request to the url https://www.test.com/sgcWebAuthn/Registration/Options.

The client sends a new request passing as a payload the following json (using test as username)

```
 {"username":"test","algorithms":[]}
```

The server reads the request and returns a response with a new challenge.

```
{
    "rp": {
        "name": "localhost",
        "id": "localhost"
    },
    "user": {
        "id": "36b9d6a84204487382fee62e7e67a80d",
        "name": "test",
        "displayName": "test"
    },
    "challenge": "6c6c468c99f24bf29a85a15b661f75f385654f97309c46bab2909c926e17ccbe",
    "pubKeyCredParams": [
        {
            "type": "public-key",
            "alg": "-7"
        },
        {
            "type": "public-key",
            "alg": "-257"
        }
    ],
    "timeout": 60000,
    "excludeCredentials": [],
    "authenticatorSelection": {
        "residentKey": "preferred",
        "requireResidentKey": false,
        "userVerification": "preferred"
    },
    "attestation": "direct",
    "hints": [],
    "extensions": {
        "credProps": true
    }
}
```

The response returns the following data:

- **challenge:** Randomized binary data, base64-encoded.
- **rp:** Relying Party info (your web app/site).
- **user:** User identifier and display name.
- **pubKeyCredParams:** Allowed algorithms (e.g., ES256, RS256).

- **authenticatorSelection:** Preferences for authenticator type (platform or cross-platform, user verification options).
- **timeout:** Suggested timeout value.
- **attestation:** Attestation conveyance preference ("none", "direct", "indirect").

Registration Verify

Now the client has the response from the server, reads the response and the authenticator returns the cryptographic data to the client web-browser. Now the client sends a new HTTP Request with the following data to the Registration Verify Endpoint configured in the TsgcWSAPIServer_WebAuthn server, by default is /sgcWebAuthn/Registration/Verify
, so if your server is listening in the domain https://www.test.com, the client should make a new request to the url https://www.test.com/sgcWebAuthn/Registration/Verify.

The authenticator responds back to the JavaScript in the browser with:

- **id:** Credential ID (unique ID for the authenticator-generated key pair).
- **rawId:** Binary form of Credential ID.
- **type:** Credential type ("public-key").
- **response:** Contains attestationObject and clientDataJSON.

Find below an example:

```
{
   "id": "yeA4BVRlrAfLG-KzqsL_rII4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
   "rawId": "yeA4BVRlrAfLG-KzqsL_rII4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
   "response": {
      "attestationObject": "o2NmbXRmcGFja2VkZ2F0dFN0....",
      "clientDataJSON": "eyJ0eXBlIjoid2ViYXV0aG4uY3Jl....",
      "transports": [
         "nfc",
         "usb"
      ],
      "publicKeyAlgorithm": -7,
      "publicKey": "MFkwEwYHKoZIzj0CAQYIKoZIzj....",
      "authenticatorData": "SZYN5YgOjGh0NBcPZHZgW4_k...."
   },
   "type": "public-key",
   "clientExtensionResults": {
      "credProps": {
         "rk": true
      }
   },
   "authenticatorAttachment": "cross-platform"
}
```

The server reads the JSON request from the client and decodes, verifies, and stores the public key and credential ID.

Decode attestationObject and clientDataJSON:

- Extract the public key from attestationObject.
- Validate the challenge in clientDataJSON matches the original one sent by the server.
- Ensure proper origin validation (myapp.example.com).

Store Credential:

- If the verification is successful, the event **OnWebAuthnRegistrationSuccessful** is called so you can store the credential ID and extracted public key securely in your database for future authentications.

- If there is any error, the event **OnWebAuthnRegistrationError** is called and you can obtain the details of the error.

Registration Flow

Get more info about the Registration Flow process using the following links:

- **WebAuthn Registration Request**
- **WebAuthn Registration Response**
- **WebAuthn Registration Result**

WebAuthn Registration | Request

This registration options request is essential to bootstrap WebAuthn registration securely. It asks the server to:

- Generate a challenge,
- Look up or provision the user ID,
- Return a valid PublicKeyCredentialCreationOptions object (with RP, user, algorithms, etc.).

The browser must obtain credential creation options from the server to initiate a secure WebAuthn credential registration. This is typically done via an HTTPS POST request from the browser to a server endpoint (e.g., /sgcWebAuthn/Registration/Options), which prepares parameters like the challenge, RP ID, and user info.

The request body contains data that identifies the user and possibly provides configuration preferences. It often looks like this:

```
{
  "username": "alice@example.com",
  "displayName": "Alice Smith",
  "authenticatorSelection": {
    "authenticatorAttachment": "platform",       // optional: platform or cross-platform
    "userVerification": "preferred"               // optional: required | preferred | discouraged
  },
  "attestation": "none",                          // or "direct", "indirect"
  "residentKey": "discouraged"                    // optional
}
```

The WebAuthn Server Component listens on the endpoint configured in the property **EndpointOptions.RegistrationOptions** the initial browser request to obtain a PublicKeyCredentialCreationOptions. When the server receives a new HTTP Request, the event **OnWebAuthnRegistrationOptionsRequest** is called and you can access the request sent by the client and cancel the request by setting the parameter Accept to False.

```
void __fastcall TForm1::OnWebAuthnRegistrationOptionsRequest(TObject *Sender, TsgcWebAuthn_RegistrationOptions_Re
{
    if (aRequest->Username == "anonymous")
        Accept = false;
}
```

WebAuthn Registration | Response

The server responds to the client's registration options request (e.g., POST /sgcWebAuthn/Registration/Options) with a JSON payload that looks like the following (after base64url-encoding binary fields):

```
{
  "publicKey": {
    "rp": {
      "name": "esegece software",
      "id": "esegece.com"
    },
    "user": {
      "id": "c3ViamVjdC1pZA",             // base64url-encoded ArrayBuffer (user han-
dle)
      "name": "webauthn@esegece.com",
      "displayName": "Delphi Developer"
    },
    "challenge": "Xz8x2K6nY3gZ...",          // base64url-encoded challenge
    "pubKeyCredParams": [
      { "type": "public-key", "alg": -7 }, // ES256
      { "type": "public-key", "alg": -257 } // RS256
    ],
    "timeout": 60000,
    "attestation": "none",
    "authenticatorSelection": {
      "authenticatorAttachment": "platform",
      "userVerification": "preferred",
      "residentKey": "discouraged"
    }
  }
}
```

Find below a description of the fields:

- **rp:** Relying Party info:
  - **name:** Friendly display name (e.g., "Example Inc.").
  - **id:** Relying Party ID (must match site origin or be a registrable suffix).
- **user:** Information identifying the user account:
  - **id:** Unique, stable byte array (e.g., user UUID).
  - **name:** Unique user identifier (username or email).
  - **displayName:** User-facing display name.
- **challenge:** A cryptographically random nonce (16–64 bytes), base64url-encoded.
- **pubKeyCredParams:** Algorithms the RP is willing to accept:
  - **Common:** -7 (ES256), -257 (RS256)
- **timeout:** Optional timeout for the creation operation (in ms).
- **attestation:** One of "none", "indirect", "direct".
- **authenticatorSelection:**
  - **authenticatorAttachment:** "platform" (built-in) or "cross-platform" (e.g., YubiKey).
  - **userVerification:** "required", "preferred", or "discouraged".
  - **residentKey:** "required", "preferred", or "discouraged".

Before the response is sent to the client, the event **OnWebAuthnRegistrationOptionsResponse** is called, allowing you to customize the response.

```
void __fastcall OnWebAuthnRegistrationOptionsResponse(TObject *Sender,
  TsgcWebAuthn_RegistrationOptions_Request* aRequest,
  TsgcWebAuthn_RegistrationOptions_Response* aResponse)
{
```

```
  if (aRequest->Username == "esegece.com") {
    aResponse->ExcludeCredentials->AddCredentialRecordFromJSON("json1.txt");
    aResponse->ExcludeCredentials->AddCredentialRecordFromJSON("json2.txt");
  }
}
```

WebAuthn Registration | Result

The goal is to verify the authenticity and integrity of the data returned by the client, ensure that the credential is bound to the expected user, and safely register a public key credential for future authentication.

The server must validate the client response following these steps:

- Parse the client response: The client sends a response like this to the server.

```
{
  "id": "base64url-encoded credential ID",
  "rawId": "base64url-encoded ID bytes",
  "response": {
    "clientDataJSON": "base64url",
    "attestationObject": "base64url"
  },
  "type": "public-key"
}
```

- The server validates the clientDataJSON and attestationObject
- Verifies the Attestation Statement (if defined)
- If everything is valid, it stores the credential and can trust the public key for future logins.

Registration Successful

If the response sent by the client is valid, the event **OnWebAuthnRegistrationSuccessful** is called and the Credential Record can be safely stored into a database for future logins validations.

```
void __fastcall OnWebAuthnRegistrationSuccessful(TObject *Sender,
  TsgcWebAuthn_Registration* aRegistration,
  TsgcWebAuthn_CredentialRecord* aCredentialRecord,
  bool &Accept)
{
  // Store in a DB
  DB->Credentials->Append();
  DB->Credentials->FieldByName("Credentials")->AsString = aCredentialRecord->AsJSON;
  DB->Credentials->Post();
}
```

Registration Error

If there is any error while validating the client response, the event **OnWebAuthnRegistrationError** is called and you can access the reason for the error in the parameter aError.

```
void __fastcall OnWebAuthnRegistrationError(TObject *Sender,
  TsgcWebAuthn_RegistrationVerify_Request* aRequest,
  TsgcWebAuthn_Registration* aRegistration,
  const String aError)
{
  Log("#webauthn_registration_error: " + aError);
}
```

WebAuthn | Authentication

WebAuthn Authentication allows users to log in using previously registered public-key credentials. It involves validating a signed challenge using the user's stored public key from registration.

Authentication Options

Authenticating requires the client starting the authentication flow using a new HTTP Request to the **Authentication Options Endpoint** configured in the TsgcWSAPIServer_WebAuthn server. By default, the endpoint is /sgcWebAuthn/Authentication/Options, so if your server is listening in the domain https://www.test.com, the client should make a new request to the url https://www.test.com/sgcWebAuthn/Authentication/Options.

Client sends the assertion (authentication response) to the Server via POST

```
{"username":"test","user_verification":"preferred"}
```

The server Generates PublicKeyCredentialRequestOptions

```
{
    "challenge": "9d0d61edf30b45f8b88aef7087f9117716e2b7d8b0ee4460b06142f39dd0ec9f",
    "timeout": 60000,
    "rpId": "localhost",
    "allowCredentials": [
        {
            "id": "yeA4BVRlrAfLG-KzqsL_rlI4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
            "type": "public-key",
            "transports": [
                "nfc",
                "usb"
            ]
        }
    ],
    "userVerification": "preferred",
    "hints": [],
    "attestation": "none",
    "attestationFormats": [],
    "extensions": {}
}
```

Authentication Verify

Authenticating requires the client starting the authentication flow using a new HTTP Request to the **Authentication Verify Endpoint** configured in the TsgcWSAPIServer_WebAuthn server. By default, the endpoint is /sgcWebAuthn/Authentication/Verify, so if your server is listening in the domain https://www.test.com, the client should make a new request to the url https://www.test.com/sgcWebAuthn/Authentication/Verify.

The browser prompts the user to use their authenticator (e.g., fingerprint, YubiKey). The authenticator signs the challenge with the private key linked to the credential ID. Returned credential includes:

- **rawId:** Credential ID (base64url).
- **response.authenticatorData:** Metadata from the authenticator (binary).
- **response.clientDataJSON:** Contains the original challenge, origin, and type.
- **response.signature:** Signature over authenticatorData + hash(clientDataJSON).

Find below a json example of the client request:

```
{
    "id": "yeA4BVRlrAfLG-KzqsL_rlI4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
```

```
    "rawId": "yeA4BVRlrAfLG-KzqsL_rlI4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
    "response": {
        "authenticatorData": "SZYN5YgOjGh0NBcPZHZgW4_krrmihjLHmVzzuoMdl2MFAAAABw",
        "clientDataJSON": "eyJ0eXBlIjoid2ViYXV0aG4uZ.....",
        "signature": "MEQCIAJRqvvys8....",
        "userHandle": "36b9d6a84204487382fee62e7e67a80d"
    },
    "type": "public-key",
    "clientExtensionResults": {},
    "authenticatorAttachment": "cross-platform"
}
```

The server reads the request from the client, validates that the credential is stored, and verifies the signature. If the signature is valid, the event **OnWebAuthnAuthenticationSuccessful** is called.

Authentication Flow

Get more info about the Authentication Flow process using the following links:

- **WebAuthn Authentication Request**
- **WebAuthn Authentication Response**
- **WebAuthn Authentication Result**

WebAuthn Authentication | Request

When a user attempts to log in, the browser sends a request to the server asking for the authentication options (also called "assertion options").

```
{
  "username": "alice@example.com"
}
```

This request allows the server to:

- Generate a challenge (a secure, random value).
- Look up which credentials (public key IDs) are valid for this user.
- Respond with cryptographic parameters that the browser will use in navigator.credentials.get().

When the WebAuthn Server Component receives this request, this is typically done via an HTTPS POST request from the browser to a server endpoint (e.g., /sgcWebAuthn/Authentication/Options), the event **OnWebAuthnAuthenticationOptionsRequest** is called, so you can add the credentials associated with the username (if any).

```
void __fastcall TForm1::OnWebAuthnAuthenticationOptionsRequest(
  TObject *Sender,
  TsgcWebAuthn_AuthenticationOptions_Request *aRequest,
  TsgcWebAuthn_CredentialRecords &CredentialRecords,
  bool &Accept)
{
    if (UserExistsInDB(aRequest->Username)) {
        while (!EOF()) {
            CredentialRecords.AddCredentialRecordFromJSON(RecordFromDB());
            Next();
        }
    }
}
```

WebAuthn Authentication | Response

Once the browser (client) sends a request to the server with the user's username (or a similar identifier), the server replies with the authentication options that the client will use to begin the authentication process via the browser's navigator.credentials.get() API.

This response provides the parameters and constraints needed by the browser to generate a WebAuthn authentication assertion using the user's authenticator (e.g., security key, biometric device).

Example JSON Response:

```
{
  "challenge": "Z3lVbWV5YXBpbmdvZG90IQ",
  "timeout": 60000,
  "rpId": "example.com",
  "allowCredentials": [
    {
      "type": "public-key",
      "id": "dXNlckNyZWRJZA",
      "transports": ["usb", "nfc", "ble"]
    }
  ],
  "userVerification": "preferred"
}
```

Find below a description of the fields:

- **challenge:** A cryptographic challenge (randomly generated) to prevent replay attacks. Must be unique per request.
- **timeout:** Optional. How long the browser should wait (in ms) before canceling the operation.
- **rpId:** Relying Party Identifier — usually your domain name.
- **allowCredentials:** A list of acceptable credentials (credential IDs) that the user previously registered.
- **userVerification:** Informs the browser how to verify the user: "required", "preferred", or "discouraged".
- **extensions (optional):** Additional capabilities or data requested from the authenticator.

Before the response is sent to the client, the event **OnWebAuthnAuthenticationOptionsResponse** is called, allowing you to customize the response.

WebAuthn Authentication | Result

The goal is to validate the signed assertion provided by the browser, which proves the user owns the private key originally registered. This is what securely logs the user in. After the user interacts with their authenticator (e.g., fingerprint, security key), the browser sends a POST request back to the server with the authentication result. Find below a json example:

```
{
  "id": "credential-id",
  "rawId": "base64url-encoded-credential-id",
  "type": "public-key",
  "response": {
    "clientDataJSON": "base64url",
    "authenticatorData": "base64url",
    "signature": "base64url",
    "userHandle": "optional"
  }
}
```

When the server receives this request at the configured endpoint (e.g., /sgcWebAuthn/Authentication/Verify), it must validate the following steps:

- Retrieve Credential from DB: Use the credential-id to lookup the user's registered public key and metadata (e.g., signature counter).
- Decode & Parse clientDataJSON
- Validate Authenticator Data
- Verify Signature

If all validations are correct, the authentication is successful and the event **OnWebAuthnAuthenticationSuccessful** is called.

If any check fails, the event **OnWebAuthnAuthenticationError** is called with the reason for the error.

WebAuthn | MDS

The **Metadata Service (MDS)** is a **centralized service provided by the FIDO Alliance** that aggregates and publishes **Metadata Statements** about authenticators certified through FIDO certification programs. These statements contain detailed security, compliance, and operational information about the authenticators.

- The service endpoint is often referred to as **MDS3** (the third version of the Metadata Service protocol).
- Relying Parties (e.g., websites or applications implementing WebAuthn) **retrieve metadata statements** from MDS to make informed trust decisions about authenticators.

The MDS adds a crucial layer of trust and security validation for relying parties using WebAuthn:

- **Authenticator Validation:** Enables verification of authenticator compliance with FIDO standards and helps validate the AAGUID presented in a WebAuthn attestation.
- **Compromise & Revocation Detection:** Provides up-to-date information on compromised or revoked authenticators, allowing relying parties to block insecure devices.
- **Security Assurance:** Helps enforce security policies, such as only allowing authenticators that meet a certain FIDO certification level or user verification strength.
- **Interoperability:** Ensures consistent behavior and security expectations across different browsers, platforms, and devices using different authenticators.

Configuration

You can configure the use of the MDS using the property WebAuthnOptions.MDS, find below the main properties:

- **Enabled:** if true (the default value), the webauthn requests will be validated against the configured MDS file.
- **MDS_FileName:** the path where the MDS file is stored. It can be downloaded from the following URL: https://mds3.fidoalliance.org/
- **RootCert_FileName:** the path where the Root Certificate is stored. Must be defined to validate the certificate chain. It can be downloaded from https://valid.r3.roots.globalsign.com/
- **Leaf_CertificateCRL:** if true (by default is false), the leaf certificate of the blob will be validated against the CRL (Certificate Revocation List).
- **CRL_FileName:** the path where the CRL file is stored. If it's not defined and Leaf_CertificateCRL is enabled, it will try to download the CRL automatically.

WebAuthn Authorization

If you want the server to send a bearer token after a successful authentication that can be used to open a new WebSocket or HTTP connection, pass the parameter token = true. Example:

```
{
  "username": "alice@example.com", "token": true
}
```

After a successful Authentication, the server will send a response like this:

```
{
  "verified": "ok",
  "authentication": {
    "token": "C760C1C39E3D4E829693A13F18F5CFDE537B516336FC48F7BAB0276176F9E6DE"
  }
}
```

The event **OnWebAuthnUnauthorized** is called when a request is not authorized and the connection will be disconnected. Here you can configure which endpoints require WebAuthn authentication and which do not.

WebAuthn Authorization | HTTP

Once you have a new token, just send an authorization header with this bearer token. You can use the Custom-Headers property of the TsgcHTTP1Client to configure the Bearer Token. Example:

```
String GetHTTPRequest(const String aURL, const String aToken)
{
    // Create the HTTP client dynamically
    std::unique_ptr<TsgcHTTP1Client> oHTTP(new TsgcHTTP1Client(nullptr));
    // Add Authorization header
    oHTTP->Request->CustomHeaders->AddValue("Authorization", "Bearer " + aToken);
    // Perform GET request and return result
    return oHTTP->Get(aURL);
}
```

WebAuthn Authorization | WebSocket

Once you have a new token, just send an authorization header with this bearer token. You can use the event On-HandShake to add the Bearer Token to the connection request. Example:

```
void __fastcall OnClientHandshake(TsgcWSConnection *Connection, TStringList *Headers)
{
    Headers->Add("Authorization: Bearer C760C1C39E3D4E829693A13F18F5CFDE537B516336FC48F7BAB0276176F9E6DE");
}
```

Webauthn | Javascript Client

WebAuthn (Web Authentication API) is a W3C standard that enables secure passwordless authentication using **public-key cryptography**. Instead of passwords, users register and authenticate using **hardware-based authenticators** (like fingerprint readers, Face ID, YubiKeys, etc.) or platform authenticators (built-in, like Touch ID).

Find below how to handle the Registration and Authentication using a Javascript client.

WebAuthn Registration

How WebAuthn Registration works

- **User Initiates Registration**:
  - The user provides a username and clicks **Register**.
- **Browser Requests Options from Server**:
  - The frontend makes a `POST` request to get registration options.
- **Browser Creates Credentials**:
  - Using the `navigator.credentials.create()` API via `SimpleWebAuthnBrowser.startRegistration()`, a credential is created.
- **Server Verifies Registration**:
  - The browser sends back the credential to the server.
  - The server verifies the registration and stores the public key for that user.

The **TsgcWSAPIServer_WebAuthn** component has an html file to test the WebAuthn protocol. This HTML file contains a minimal UI and JavaScript to interact with WebAuthn.

Walkthrough of sgcWebAuthn.html

Structure Overview:

- **Username Input**: Captures the user's identifier.
- **Buttons**:
  - `Register` – initiates WebAuthn registration.
  - `Authenticate` – initiates login (handled similarly).
- **Debug Console**: Shows real-time debug information (JSON from WebAuthn).

1. HTML UI for Input

```
<input type="text" id="username" name="username" autocomplete="username webauthn" />
<button id="btnRegBegin"><strong>Register</strong></button>
```

2. JavaScript: Button Click Handler

```
document.querySelector('#btnRegBegin').addEventListener('click', async () => {
  const username = document.getElementById("username").value;
  if (username == "") {
    document.getElementById('Error').innerText = 'Please enter a username to register';
    return;
}
  const resp = await fetch('<#webauthn_registration_options>', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ username, algorithms: [] })
});
  const options = await resp.json();
const attResp = await startRegistration(options);  // WebAuthn API
```

3. Server Response (Fake Endpoint in HTML)

```
fetch('/sgcWebAuthn/Registration/Options', ...)
fetch('/sgcWebAuthn/Registration/Verify', ...)
```

4. Finalizing Registration

```
const verificationResp = await fetch('/webauthn/register/verify', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(attResp)
});
const verificationJSON = await verificationResp.json();
if (verificationJSON && verificationJSON.verified) {
  document.getElementById('Success').innerHTML = `Authenticator registered!`;
}
```

WebAuthn Authentication

How WebAuthn Authentication works

- **User Initiates Authentication**:
  - The user provides a username and clicks **Register**.
- **Browser Requests Options from Server**:
  - The frontend makes a `POST` request to get authentication options.
- **Browser Creates Credentials**:
  - Using the `navigator.credentials.get()` API via `SimpleWebAuthnBrowser.startAuthentication()`.
- **Server Verifies Authentication**:
  - The browser sends back the credential to the server.
  - The server verifies the signed result.

1. HTML UI Setup

```
<div class="container">
<h1>WebAuthn Authentication Sample</h1>
  <section id="userdata">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" autocomplete="username webauthn" autofocus />
</section>
  <button id="btnAuthBegin"><strong>Authenticate</strong></button>
  <p id="Success" class="success"></p>
<p id="Error" class="error"></p>
  <details open>
    <summary>Console</summary>
    <textarea id="Debug"></textarea>
  </details>
</div>
```

2. Get Authentication Options

Before calling startAuthentication, you send the username to the server

```
const resp = await fetch('<#webauthn_authentication_options>', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    username: document.getElementById("username").value,
    user_verification: 'preferred'
  }),
});
```

The server responds with a JSON object that includes:

```
{
  "challenge": "base64url-encoded-random-string",
  "allowCredentials": [
    {
      "id": "base64url-credential-id",
      "type": "public-key"
    }
  ],
  "userVerification": "preferred",
  "rpId": "yourdomain.com"
}
```

This is called the PublicKeyCredentialRequestOptions.

3. Receive the Authenticator Response

The asseResp looks like this (simplified):

```
{
  "id": "credentialId",
  "rawId": "base64url-encoded-id",
  "response": {
    "authenticatorData": "...",
    "clientDataJSON": "...",
    "signature": "...",
    "userHandle": "..."
  },
  "type": "public-key",
  "clientExtensionResults": {}
}
```

This response proves that the user:

- Possesses the private key stored on the authenticator
- Signed the server's challenge
- Was physically present (if required)

4. Send the Signed Authentication Response to the Server

After the user interacts with their authenticator (via startAuthentication()), you get a response object in JavaScript.

```
const verificationResp = await fetch('/webauthn/authenticate/verify', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(asseResp), // this is the signed response
});
```

5. Get the Server's Response

The server will reply with a result like this:

```
{ "verified": true }
```

Or if something went wrong:

```
{ "verified": false, "error": "Invalid signature" }
```

And you handle it in your frontend code:

const result = await verificationResp.json();

```
if (result.verified) {
  document.getElementById('Success').textContent = 'User authenticated!';
} else {
  document.getElementById('Error').textContent = 'Authentication failed!';
}
```

TsgcTOTPAuthenticator

TOTP (RFC 6238) and HOTP (RFC 4226) one-time password component for two-factor authentication.

Introduction

The **TsgcTOTPAuthenticator** component implements time-based one-time passwords (**TOTP**, RFC 6238) and HMAC-based one-time passwords (**HOTP**, RFC 4226), the codes shown by authenticator apps such as Google Authenticator, Microsoft Authenticator or Authy. Use it to add a second authentication factor to a login form, a REST API or a WebSocket server.
The component is non-visual and never opens a connection: it creates secrets, builds the `otpauth://` provisioning URI shown as a QR code, generates and verifies codes, and creates one-time recovery codes. Storing the secrets is the job of the application.

Availability

TsgcTOTPAuthenticator is included in **sgcWebSockets Enterprise** and in the **sgcAuth** pack. The unit is `sgcAuth_TOTP`.

How TOTP works

1. The server creates a random secret for every user and shares it once with the authenticator app, usually as a QR code.
2. Both sides divide the current Unix time by **Period** (30 seconds) to get the same time step counter.
3. Both sides compute the HMAC (SHA-1 by default) of that counter keyed with the secret.
4. A dynamic truncation turns the HMAC into a decimal code of **Digits** digits (6 by default).
5. The server accepts the code when it matches the current time step or one of the **Window** steps before or after it.

Quick Start: enrolment

Create a secret, show the provisioning URI as a QR code and confirm the enrolment with a first code typed by the user. Save the secret only when the first code is valid.

Quick Start: verification with replay protection

A TOTP code stays valid for the whole time step, and with **Window** = 1 for the steps before and after it. Store the last accepted time step of every user and pass it to **VerifyCode**: only a code for a later step is accepted, so a code which has already been used, or which was captured by an attacker, can not be used again.

Security notes

- **Store the secrets encrypted.** The TOTP secret works like a password shared between the server and the app. Anyone who reads it can generate valid codes, so keep it encrypted at rest and never log it.
- **Store the recovery codes hashed. GenerateRecoveryCodes** only creates the codes. Show them to the user once, save a hash of every code (for example SHA-256), compare the hash when a code is used and delete it after a successful login, because every recovery code is valid only once.
- **Persist the last time step.** Replay protection only works when the value returned in `aLastTimeStep` is saved for the user after every successful verification.
- **Limit the attempts.** A 6 digit code has one million values, lock the account or add a delay after a few failed codes.
- **Delphi 7.** On Delphi 7 HMAC-SHA256 and HMAC-SHA512 are computed by OpenSSL. When **Algorithm** is `totpSHA256` or `totpSHA512`, deploy the OpenSSL 3 libraries with the application and keep OpenSSL 3 selected (`OPENSSL_API_VERSION := opSSL_3_0`, the default value in `sgcIdSSLOpenSSLHeaders`).

Reference

- Properties: 7 properties available on the component.
- Methods: 7 public methods.

TsgcLDAPClient

LDAP v3 and Active Directory client component: authentication, search with paging and group membership.

Introduction

The **TsgcLDAPClient** component is an LDAP v3 client (RFC 4511, RFC 4513) designed to authenticate users against **Active Directory**, OpenLDAP or any other LDAP directory, search the directory and read the groups of a user. It uses Indy for the transport and the OpenSSL IO handler for TLS.
Only one operation runs at a time, every public method is serialized, so a single instance can be shared by several threads.

Availability

TsgcLDAPClient is included in **sgcWebSockets Enterprise** and in the **sgcAuth** pack. The unit is `sgcAuth_LDAP_Client`, the protocol types and helper functions are declared in `sgcAuth_LDAP_Classes`.

Security modes

The **Security** property selects how the connection is protected:

- `ldapsecNone`: plain LDAP on port 389. Passwords travel in clear text, use it only on a trusted network or for tests.
- `ldapsecLDAPS`: implicit TLS, the TLS handshake starts as soon as the TCP connection is open. Selecting it changes **Port** from 389 to 636.
- `ldapsecStartTLS`: the client connects to port 389 and upgrades the connection with the StartTLS extended operation (RFC 4511 section 4.14) before anything else is sent. If the server refuses StartTLS, **Connect** raises an exception and the connection is closed, the client never continues in clear text.

The TLS settings are in **TLSOptions**. **TLSOptions.VerifyCertificate** is **True** by default: the certificate chain of the server is verified by OpenSSL against the CA certificates of **TLSOptions.RootCertFile**, and with the sgcWebSockets Indy library the certificate must also match **Host**. For Active Directory set **RootCertFile** to a PEM file with the root certificate of your enterprise CA. When the handshake fails, the exception message includes the OpenSSL verification error, for example "unable to get local issuer certificate". Only the OpenSSL IO handler is supported.

Active Directory authentication

Authenticate validates the credentials of a user with a simple bind. **AuthenticationMode** selects how the user name is turned into the bind name. After every call the connection is bound again with **BindDN** and **Password** (the service account, or an anonymous bind when both are empty), so the connection never stays bound as the user.
UPN (`ldapamUPN`): the user signs in with a user principal name such as `john@contoso.com`. When the name has no `@`, **UPNSuffix** is appended.
DownLevel (`ldapamDownLevel`): the classic `DOMAIN\user` logon name. When the name has no `\`, **Domain** is prepended.
SearchThenBind (`ldapamSearchThenBind`, the default): the client binds with the service account (**BindDN**, **Password**), searches **UserSearchFilter** under **BaseDN** and binds with the DN it found. The search must return exactly one entry. This mode works with any directory and lets the user sign in with the `sAMAccountName`, or any attribute you put in the filter.
DN (`ldapamDN`): the user name is already the full distinguished name.
With UPN and DownLevel, when **BaseDN** is set, the DN of the user entry is searched after the bind and returned in `aUserDN`. Otherwise `aUserDN` returns the bind name.

Empty passwords are rejected

RFC 4513 section 5.1.2 defines a bind with a name and an empty password as an *unauthenticated bind*, and many servers answer it with success. A login form which forwards an empty password would accept any user name. **Authenticate** and **Bind** refuse a name with an empty password without contacting the server: they return False with

LastResultCode = 48 (inappropriateAuthentication). An empty DN with an empty password is still an anonymous **Bind**.

Search with paging

Search returns the entries in a **TsgcLDAPEntries** list. When **PageSize** is greater than 0 (500 by default) the client uses the Simple Paged Results control (RFC 2696) and requests the next pages automatically, so searches return more entries than the page limit of the server (1000 in Active Directory). Continuation references and referrals are collected in **Referrals**, they are never followed.

Groups

GetUserGroups returns the DNs of the groups of a user. Without nesting it reads the `memberOf` attribute of the user entry, which only lists the direct groups. With `aNested` = True it searches under **BaseDN** with the Active Directory matching rule `LDAP_MATCHING_RULE_IN_CHAIN` (`1.2.840.113556.1.4.1941`), which also returns the groups the user belongs to through other groups: `(member:1.2.840.113556.1.4.1941:=<user DN>)`.

Escaping helpers against LDAP injection

Never concatenate user input into a filter or a DN. The unit `sgcAuth_LDAP_Classes` declares the escaping functions:

- `sgcLDAPEscapeFilterValue(const aValue: string): string`: escapes `*`, `(`, `)`, `\` and NUL of a filter assertion value (RFC 4515).
- `sgcLDAPEscapeDNValue(const aValue: string): string`: escapes an attribute value used inside a DN (RFC 4514).

Authenticate and **GetUserGroups** already escape the values they put in their filters. Use the helpers for your own **Search** calls.

Error handling

A result returned by the server is stored in **LastResultCode**, **LastErrorMessage** (the diagnostic message of the server, or the RFC 4511 name of the code when the server sent none) and **LastMatchedDN**. `sgcLDAPResultCodeToText` converts a code to its name, for example 49 to `invalidCredentials`.

- **Bind** and **Authenticate** return False when the credentials are refused.
- Other server errors, protocol errors, timeouts and TLS failures raise **EsgcLDAPError**, and the **OnError** event is fired with the message. A server side error keeps the connection open, a transport or protocol error closes it.
- An invalid filter raises **EsgcLDAPFilterError**, whose **Position** property is the position of the error in the filter, before anything is sent.

Active Directory returns code 49 for every refused bind and explains the reason in the diagnostic message with a `data` value, for example `80090308: LdapErr: DSID-0C09044E, comment: AcceptSecurityContext error, data 52e, v4563`. The component returns that text unchanged in **LastErrorMessage**, the most common values are:

- `525`: user not found.
- `52e`: invalid credentials (wrong password).
- `530`: logon not permitted at this time. `531`: logon not permitted at this workstation.
- `532`: password expired. `773`: the user must reset the password.
- `533`: account disabled. `701`: account expired. `775`: account locked out.

Do not show these details to the user who is signing in, a generic "invalid user name or password" message does not reveal which accounts exist. Log them for the administrators.

Reference

- Properties: 20 properties available on the component.
- Methods: 8 public methods.
- Events: 3 events you can handle.

TsgcSAMLServiceProvider

SAML 2.0 Web Browser SSO service provider: AuthnRequest, SP metadata and validation of the signed IdP response.

Introduction

TsgcSAMLServiceProvider adds SAML 2.0 single sign-on to your web applications. It implements the service provider (SP) side of the SAML 2.0 Web Browser SSO profile: it creates the **AuthnRequest** sent to the identity provider (IdP), publishes the service provider metadata and validates the signed **Response** that the browser posts back to the Assertion Consumer Service (ACS) URL.
The component is not an HTTP server. Host the login, ACS and metadata URLs in any HTTP server, for example **TsgcWebSocketHTTPServer** or a **TsgcHTTPServer**, and call the component from the request handler. It works with Microsoft Entra ID, Okta, AD FS, Google Workspace, Keycloak and any other SAML 2.0 identity provider.

Availability

TsgcSAMLServiceProvider is included in **sgcWebSockets Enterprise** and in the **sgcAuth** pack. The unit is `sgcAuth_SAML_SP`. The validation result is returned in a TsgcSAMLResult object.

SP-initiated sign in, step by step

1. **Configure the service provider.** Set **EntityID** (the unique name of your application, usually the metadata URL) and **AssertionConsumerServiceURL** (the https URL where the IdP posts the response). Serve **GetMetadata** and register it in the identity provider.
2. **Configure the identity provider.** Call **LoadIdPMetadata** with the IdP metadata, or set **IdPEntityID**, **IdPSSOURL** and **IdPCertificates** by hand.
3. **Send the AuthnRequest.** When an anonymous user opens the login URL, call **GetAuthnRequestRedirectURL** and answer with a 302 redirect (HTTP-Redirect binding), or return the HTML page of **GetAuthnRequestPostForm**, which posts the request to the IdP (HTTP-POST binding).
4. **Keep the request id.** Both methods return the id of the new AuthnRequest in `aRequestID`. Store it on the server, for example in a list keyed by a random RelayState value or by the session cookie. Remove it when the response arrives, so every request can be answered only once.
5. **Process the response at the ACS URL.** The browser posts the form fields `SAMLResponse` and `RelayState`. Look up the stored request id and call **ProcessResponse**. When it returns True the user is authenticated: read **NameID**, **SessionIndex** and **Attributes** from the TsgcSAMLResult and create your own session. When it returns False, **ErrorMessage** tells why and **OnSAMLError** fires.

The RelayState is not covered by the signature of the IdP. Use it only as a key to find your own pending request, never as a URL to redirect to without checking it.

Configure the identity provider

The easiest way is the metadata document of the IdP. **LoadIdPMetadata** reads the entity ID, the `SingleSignOnService` URL (HTTP-Redirect preferred, HTTP-POST otherwise) and every signing certificate, and fills **IdPEntityID**, **IdPSSOURL**, **IdPSSOBinding** and **IdPCertificates**. Download the metadata over https with any HTTP client, or save it to a file. When the document is an `EntitiesDescriptor` with several entities, pass the entity ID of the IdP in the second parameter.
Without metadata, set the same properties by hand with the values shown in the IdP console:
Notes for the most common identity providers. In all of them the two values to register are the **SP Entity ID** (**EntityID**) and the **ACS URL** (**AssertionConsumerServiceURL**). Keep assertion encryption disabled, encrypted assertions are not supported.

- **Microsoft Entra ID.** Enterprise applications, New application, Create your own application (non-gallery). In Single sign-on choose SAML and upload the SP metadata, or fill Identifier (Entity ID) and Reply URL (Assertion Consumer Service URL). Assign users or groups. Load the *App Federation Metadata Url* shown in SAML Certificates. Entra ID signs the assertion with RSA-SHA256 by default. The attributes use claim URIs as names, for example `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress`.

- **Okta.** Applications, Create App Integration, SAML 2.0. Single sign-on URL is the ACS URL (keep "Use this for Recipient URL and Destination URL" checked) and Audience URI is the SP Entity ID. Choose the Name ID format and add attribute statements such as email, firstName and lastName. Assign people or groups and load the Metadata URL of the Sign On tab.
- **AD FS.** Add a claims aware Relying Party Trust and import the SP metadata file, or enter the SAML 2.0 WebSSO URL (the ACS URL) and the relying party identifier (the SP Entity ID) by hand. AD FS only accepts https endpoints. Add a claim rule that sends a Name ID, for example "Send LDAP Attributes as Claims" (E-Mail-Addresses to E-Mail Address) followed by "Transform an Incoming Claim" (E-Mail Address to Name ID). The IdP metadata is at `https://<adfs-host>/FederationMetadata/2007-06/FederationMetadata.xml`.
- **Google Workspace.** In the Admin console open Apps, Web and mobile apps, Add app, Add custom SAML app. Download the IdP metadata on the Google Identity Provider details page. On the Service provider details page enter the ACS URL and the Entity ID and choose the Name ID format and value (for example EMAIL and Primary email). Turn the app on for the users in User access.
- **Keycloak.** Create a client of type SAML whose Client ID is the SP Entity ID (or import the SP metadata) and set the ACS URL in the valid redirect URIs or the Assertion Consumer Service POST Binding URL. Keycloak signs the whole document by default: enable *Sign assertions* too, because **WantAssertionsSigned** is True by default. When *Client signature required* is enabled, set **SignAuthnRequests**, **SPCertificate** and **SPPrivateKey**, or disable that option. The IdP metadata is at `https://<host>/realms/<realm>/protocol/saml/descriptor`.

Example: service provider hosted in an HTTP server

A complete service provider with three endpoints: `/saml/metadata` returns the SP metadata, `/saml/login` sends the AuthnRequest and `/saml/acs` validates the response. `CreateUserSession` and `WriteToLog` stand for your own code. In production remove the pending requests which have not been answered after a few minutes.

Security checks performed by ProcessResponse

A response is accepted only when every check passes. The first failure stops the validation and its reason is returned in **ErrorMessage**.

- **Safe XML parsing.** The response must be valid Base64 and valid XML. `DOCTYPE` declarations are rejected (no DTD, no external entities) and the size and nesting depth of the document are limited.
- **Status.** The root element must be a SAML 2.0 `samlp:Response` with the status `Success`. Any other status is reported with the status code and message of the IdP.
- **Signature.** The Response or the Assertion must be signed and, with **WantAssertionsSigned** (the default), the Assertion must carry its own signature. Signatures are verified only against the certificates in **IdPCertificates**: a certificate embedded in the message (`KeyInfo`) is never trusted. Supported algorithms are RSA-SHA256, RSA-SHA384 and RSA-SHA512 with exclusive XML canonicalization.
- **SHA-1 disabled by default.** RSA-SHA1 signatures and SHA-1 digests are rejected unless **AllowSHA1** is True.
- **Signature wrapping protection.** The response must contain exactly one Assertion, as a direct child of the Response. The signature must reference the ID of the signed element, that ID must be unique in the document, and after the verification only the element covered by the signature is read.
- **Issuer.** The Issuer of the Assertion, and of the Response when present, must be **IdPEntityID**.
- **Destination and recipient.** The Destination of the Response, when present, and the Recipient of the bearer `SubjectConfirmationData` must be **AssertionConsumerServiceURL**.
- **Audience.** The Assertion must have at least one `AudienceRestriction` and every one of them must contain **EntityID**.
- **Time window.** `NotBefore` and `NotOnOrAfter` of the Conditions and of the bearer confirmation, and `SessionNotOnOrAfter` of the AuthnStatement, are checked against the current UTC time with a tolerance of **ClockSkew** seconds. The bearer confirmation must have a `NotOnOrAfter`. Set **MaxAssertionAge** to also limit the age of the Assertion.
- **InResponseTo.** The Response and the bearer confirmation must answer the request id passed in `aExpectedRequestID`. Unsolicited (IdP-initiated) responses are rejected unless **AllowIdPInitiated** is True, and then they must not contain an InResponseTo.
- **Single assertion.** Responses with several assertions, with an `EncryptedAssertion` or with an encrypted NameID are rejected.
- **Replay cache.** The ID of every accepted Assertion is kept until the Assertion expires. The same Assertion posted a second time is rejected. The cache is thread safe and lives in memory in the component. When several servers share the sign in, descend from the component and override the protected virtual method `DoAddToReplayCache` to keep the IDs in a shared store.

OpenSSL 3

The XML signatures of the IdP are verified with OpenSSL, and OpenSSL signs the AuthnRequests when **SignAuthnRequests** is enabled. **OpenSSL_Options.APIVersion** is `oslAPI_3_0` by default: deploy the OpenSSL 3 libraries (`libcrypto-3.dll` and `libssl-3.dll` on Windows) with the application.

Limitations

- Encrypted assertions (`EncryptedAssertion`) and encrypted identifiers (`EncryptedID`) are not supported. Disable assertion encryption in the IdP.
- Single Logout (SLO) is not implemented. **SessionIndex** is returned so the application can implement its own logout.
- The HTTP-Artifact binding is not supported. The AuthnRequest uses the HTTP-Redirect or HTTP-POST binding and the ACS accepts the HTTP-POST binding.
- Signed AuthnRequests are only available with the HTTP-Redirect binding (RSA-SHA256).
- Only RSA signatures are verified.

Demo

The demo **Demos\26.Authentication\03.SAML_ServiceProvider** runs a complete service provider on a TsgcWebSocketHTTPServer, loads the IdP metadata from a URL or a file and signs users in with mocksaml.com, Microsoft Entra ID, Okta or AD FS.

Reference

- Properties: 16 properties available on the component.
- Methods: 6 public methods.
- Events: 1 events you can handle.

TsgcSAMLResult

Result of the validation of a SAML response: authenticated user, attributes and error message.
Result of TsgcSAMLServiceProvider.ProcessResponse. The application creates the object, passes it to **ProcessResponse** and frees it. The identity fields are only filled when the response has been fully validated. The class is declared in the unit `sgcAuth_SAML_SP`.

Properties

All the properties are read only.

Name	Description
`NameID`	Name identifier of the authenticated user, from the Subject of the Assertion.
`NameIDFormat`	Format of the NameID, for example `urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress`.
`SessionIndex`	SessionIndex of the AuthnStatement, the identifier of the session at the IdP.
`AuthnInstant`	Time of the authentication at the IdP, as written in the AuthnStatement (ISO 8601, UTC).
`Issuer`	Entity ID of the identity provider (**IdPEntityID**).
`AssertionID`	ID of the accepted Assertion, the value stored in the replay cache.
`InResponseTo`	Request ID answered by the response. Empty for an IdP-initiated response.
`Attributes`	`TStringList` with one `Name=Value` line per `AttributeValue`. A name repeats when the attribute has several values, for example several groups. `Attributes.Values[Name]` returns the first value.
`AttributeFriendlyNames`	`TStringList` with one `Name=FriendlyName` line for every attribute that declares a FriendlyName.
`RelayState`	The RelayState passed to **ProcessResponse**. Filled even when the validation fails.
`StatusCode`	Top level status code of the Response, `urn:oasis:names:tc:SAML:2.0:status:Success` when the IdP authenticated the user.
`ErrorMessage`	Reason of the failure when **ProcessResponse** returns False. Empty on success.

Methods

Name	Description
`Clear`	Empties every property. **ProcessResponse** calls it before the validation.

Example

Back to TsgcSAMLServiceProvider

TsgcHTTP_OIDC_Client

OpenID Connect client: discovery, sign in with PKCE and nonce, ID token validation and userinfo.

Introduction

TsgcHTTP_OIDC_Client signs users in with any **OpenID Connect** provider (Google, Microsoft Entra ID, Okta, Auth0, Keycloak, AWS Cognito and others) and validates the **ID token** they return. It extends the TsgcHTTP_OAuth2_Client: the authorization code flow, the loopback server, the token requests, refresh and DPoP work as in OAuth 2.0, and the component adds discovery, PKCE and nonce by default, validation of the ID token against the keys of the provider, and the userinfo endpoint.
The unit also contains TsgcOIDCJWKS, a thread safe cache of the signing keys of an issuer, and the function sgcOIDC_ValidateIDToken, which validates tokens on the server side, for example the bearer tokens received by a REST API.

Availability

TsgcHTTP_OIDC_Client is included in **sgcWebSockets Enterprise** and in the **sgcAuth** pack. The unit is `sgcAuth_OIDC_Client`. The HTTPS requests use OpenSSL, deploy the OpenSSL 3 libraries with the application.

TLS and certificate verification

The discovery, JWKS, userinfo and token requests verify the server certificate by default (**HTTPClientOptions.TLSOptions.VerifyCertificate** is `True`). With the OpenSSL IO handler, OpenSSL does not read the Windows certificate store, so set **HTTPClientOptions.TLSOptions.RootCertFile** to a PEM file with the trusted root CA certificates (for example a `cacert.pem` bundle from curl.se), or on Windows set **HTTPClientOptions.TLSOptions.IOHandler := iohSChannel** to use the Windows certificate store instead. Without one of the two, the requests fail with an error that names the failing URL and explains that OpenSSL has no trusted root CA certificates.

Discovery

Set **OIDCOptions.Issuer** and call **Discover**. The component reads `<Issuer>/.well-known/openid-configuration` over https, checks that the `issuer` of the document is the configured issuer, and sets **AuthorizationServerOptions.AuthURL**, **AuthorizationServerOptions.TokenURL**, **JWKSURI**, **UserInfoEndpoint** and **EndSessionEndpoint**. You do not need to call it before a sign in: **Start** discovers the provider when the issuer has not been discovered yet.
Typical issuers: `https://accounts.google.com`, `https://login.microsoftonline.com/<tenant-id>/v2.0`, `https://<domain>.okta.com`, `https://<tenant>.auth0.com/` and `https://<host>/realms/<realm>` for Keycloak. Type the issuer exactly as the provider publishes it, including a final slash when it has one.

Sign in with PKCE and nonce

Register the application with the provider as a desktop or native application with a loopback redirect URL, then configure the client id, the scopes and the loopback server and call **Start**:

1. **Start** discovers the provider and adds the `openid` scope when it is missing.
2. With **OIDCOptions.UsePKCE** (the default) the authorization code grant becomes authorization code with PKCE (`auth2CodePKCE`).
3. A random `nonce` is created for every sign in and added to the authorization URL, then the browser opens the sign in page of the provider.
4. The provider redirects to the loopback server with the code and the component exchanges it for the access token and the ID token.
5. With **OIDCOptions.ValidateIDToken** (the default) the ID token is validated, including the nonce, and **OnOIDCIDToken** fires. **IDToken**, **IDTokenClaims** and **IDTokenValid** keep the result.

OnOIDCIDToken

The event receives the claims of the ID token as a JSON string, whether the token is valid and the reason when it is not. Sign the user in only when `aValid` is True, and identify the user by the `sub` claim (together with the issuer), not by the email address.
The event runs in the thread of the loopback server, synchronize with the main thread before you update the user interface.

UserInfo

After the sign in, **GetUserInfo** calls the userinfo endpoint of the provider with the access token and returns its JSON response, for example the name, the picture and the email of the user. Pass an access token to use a different one.

Microsoft Entra ID multi-tenant applications

The issuers `https://login.microsoftonline.com/common/v2.0`, `.../organizations/v2.0` and `.../consumers/v2.0` do not identify one organization: their discovery document publishes the issuer as the template `https://login.microsoftonline.com/{tenantid}/v2.0`. The component accepts that template, and when it validates an ID token it requires a `tid` claim with a GUID value and an `iss` claim equal to the template filled with that `tid`.
Such a configuration accepts users of **any** Entra ID organization. Add the tenant ids of the organizations you accept to **OIDCOptions.AllowedTenants**: when the list is not empty, a token whose `tid` is not in the list is rejected with "Tenant ... is not an allowed tenant.". Personal Microsoft accounts use the tenant `9188040d-6c67-4c5b-b112-36a304b66dad`.

Server-side validation: sgcOIDC_ValidateIDToken

A REST API or a WebSocket server that receives a token in the `Authorization: Bearer` header must check its signature and claims before it trusts it. The function **sgcOIDC_ValidateIDToken** does it with the keys of a TsgcOIDCJWKS cache. It is declared in the unit `sgcAuth_OIDC_Client`, and it is the same validation used by the component for the ID tokens.

```
bool __fastcall sgcOIDC_ValidateIDToken(const UnicodeString aToken,
  TsgcOIDCJWKS* const aJWKS, const UnicodeString aIssuer,
  const UnicodeString aClientId, const UnicodeString aNonce,
  UnicodeString &aClaims, UnicodeString &aError, int aClockSkew = 120,
  System::Classes::TStrings* const aAllowedTenants = NULL);
```

Name	Type	Description
`aToken`	`const string`	The token in compact JWS format (header.payload.signature), at most 65536 characters.
`aJWKS`	`TsgcOIDCJWKS`	Key cache of the issuer. When its **JWKSURL** is set, the keys are downloaded on first use and when an unknown `kid` appears.
`aIssuer`	`const string`	Expected `iss` claim. It can contain `{tenantid}` for Entra ID multi-tenant issuers.
`aClientId`	`const string`	Expected audience: the `aud` claim must contain it.
`aNonce`	`const string`	Expected `nonce` claim. Pass an empty string when the token has no nonce to check, for example an access token.
`aClaims`	`out string`	The JSON payload of the token when it is valid, empty otherwise.
`aError`	`out string`	Reason of the failure, empty when the token is valid.
`aClockSkew`	`Integer`	Tolerance in seconds for `exp`, `nbf` and `iat`. Default 120.

aAllowed-Tenants	TStrings	Optional list of accepted tid values. Nil or empty accepts any tenant.

The function returns True when the token is valid. It never raises an exception and it can be called from several threads with the same **TsgcOIDCJWKS**. Validate only tokens issued for your application: ID tokens, or access tokens that the provider issues for your own API (the audience is then the identifier of the API).

Accepted algorithms and claim checks

- **Format.** The token must be a signed compact JWS with three Base64url parts. Headers with a `crit` parameter are rejected.
- **Algorithms.** Only `RS256`, `RS384`, `RS512`, `ES256` and `ES384` are accepted. `none` and the symmetric `HS256`, `HS384` and `HS512` are always rejected, so a token can not be forged with the public key used as an HMAC secret.
- **Key selection.** The key is chosen by the `kid` of the header and its type, algorithm and curve must match the algorithm of the token. A token without `kid` is accepted only when exactly one key matches. Keys marked for encryption (`use` other than `sig`), RSA keys shorter than 2048 bits, symmetric keys and unsupported curves are ignored.
- **Key rotation.** An unknown `kid` downloads the JWKS again, at most once every **RefetchInterval** seconds (60 by default).
- **iss.** Must be equal to the expected issuer, or match the `{tenantid}` template with the GUID of the `tid` claim.
- **tid.** Must be in the allowed tenants when the list is not empty.
- **aud and azp.** The audience must contain the client id. When it contains several values, `azp` is required and must be the client id.
- **exp.** Required, the token is rejected after `exp` plus the clock skew.
- **nbf.** Optional, the token is rejected before `nbf` minus the clock skew.
- **iat.** Required, it can not be in the future beyond the clock skew.
- **nonce.** When an expected nonce is given, the claim must be present and equal (constant time comparison).
- **Secure URLs.** The discovery document, the JWKS and the userinfo endpoint are only downloaded over https (http is only allowed for `localhost`, `127.0.0.1` and `[::1]`) and the size of the documents is limited.

Inherited members

The OAuth 2.0 members are documented in the TsgcHTTP_OAuth2_Client reference: the properties AuthorizationServerOptions, OAuth2Options, LocalServerOptions, HTTPClientOptions and DPoPOptions, the methods Start, Stop, Refresh, Revoke and Introspect, and the events OnBeforeAuthorizeCode, OnAfterAccessToken, OnErrorAccessToken and the others. A refreshed ID token is validated too, without nonce, when the token response contains one.

Demo

The demo **Demos\26.Authentication\04.OpenID_Connect** discovers Google or Microsoft Entra ID, signs in with PKCE and nonce, shows the validated claims and the userinfo response, and validates a pasted token with **sgcOIDC_ValidateIDToken**.

Reference

- Properties: 10 properties available on the component.
- Methods: 3 public methods.
- Events: 1 events you can handle.

TsgcOIDCJWKS

Thread safe cache of the signing keys (JWKS) of an OpenID Connect issuer.
Thread safe cache of the public signing keys of an OpenID Connect issuer (JSON Web Key Set, RFC 7517). It is used by TsgcHTTP_OIDC_Client (property **JWKS**) and by the function sgcOIDC_ValidateIDToken. The class is declared in the unit `sgcAuth_OIDC_Client` and is available in sgcWebSockets Enterprise and in the sgcAuth pack.
RSA keys (2048 bits or more) and EC keys on the P-256 and P-384 curves are loaded. Keys whose `use` is not `sig`, symmetric keys and other key types are ignored. When a token arrives with a `kid` that is not in the cache, the key set is downloaded again, so the cache follows the key rotation of the provider without a restart.

Properties

Name	Type	Description
`JWKSURL`	`string`	URL of the JSON Web Key Set. Set by **TsgcHTTP_OIDC_Client.Discover** or by the application. It must use https (http only for loopback addresses).
`RefetchInterval`	`Integer`	Minimum time in seconds between two downloads caused by an unknown `kid`. Default 60. A negative value downloads the keys only once.
`TLSOptions`	`TsgcHTTPTLS_Options`	TLS settings of the download, for example the OpenSSL version. It is a reference, the object is not owned by the cache. Nil uses the defaults of the HTTP client, which verify the server certificate; set **RootCertFile** to a PEM CA bundle, or use SChannel on Windows, see TLS and certificate verification.
`Count`	`Integer`	Number of usable keys in the cache. Read only.

Methods

Declaration	Description
`function FetchFromURL(const aURL: string = ''): Integer;`	Downloads the key set from `aURL` (which becomes the new **JWKSURL**) or from **JWKSURL**, replaces the cached keys and returns their number. The server certificate is verified by default, raises an exception when the download fails, the certificate cannot be verified, or the document has no `keys` array.
`function LoadFromJSON(const aJSON: string): Integer;`	Replaces the cached keys with the keys of a JWKS document and returns their number.
`function FindKey(const aKid, aAlg: string; out aPEM: string): Boolean;`	Looks for a cached key that matches the `kid` and the algorithm and returns it as a PEM public key. It never downloads.
`function GetKeyPEM(const aKid, aAlg: string; out aPEM: string; out aError: string): Boolean;`	Like **FindKey**, but downloads the key set again when the `kid` is unknown (respecting **RefetchInterval**) and returns the reason in `aError` when no key matches.
`procedure Clear;`	Removes every key, the next validation downloads the key set again.

Example

Back to TsgcHTTP_OIDC_Client

TsgcMailOAuth2

OAuth 2.0 tokens and SASL XOAUTH2 / OAUTHBEARER strings for SMTP, IMAP and POP3 with Microsoft 365 and Gmail.

Introduction

The **TsgcMailOAuth2** component obtains and refreshes the OAuth 2.0 access tokens needed to send and read mail with **Microsoft 365** (Exchange Online) and **Gmail**, and builds the SASL **XOAUTH2** and **OAUTHBEARER** (RFC 7628) strings used by SMTP, IMAP and POP3. Both providers have disabled basic authentication with a password for most accounts, so a mail client needs OAuth2 to sign in.
The component is transport agnostic, it never talks to the mail server. Use it with Indy `TIdSMTP`, `TIdIMAP4`, `TIdPOP3` or any other mail library which can send a raw SASL command.

Availability

TsgcMailOAuth2 is included in **sgcWebSockets Enterprise** and in the **sgcAuth** pack. The unit is `sgcAuth_Mail_OAuth2`.

Microsoft 365 setup

1. In the Azure portal open **Microsoft Entra ID** › **App registrations** › **New registration** and register the application.
2. In **Authentication** add the platform **Mobile and desktop applications** with the loopback redirect URI of **LocalServerOptions** (for example `http://localhost`). For the device code flow enable **Allow public client flows**.
3. In **API permissions** add the delegated permissions for the protocols you use: `SMTP.Send`, `IMAP.AccessAsUser.All`, `POP.AccessAsUser.All`, plus `offline_access`.
4. Copy the **Application (client) ID** to **ClientId**. For a single tenant application copy the **Directory (tenant) ID** to **TenantId**, otherwise keep `common`.
5. SMTP AUTH must be enabled for the mailbox, for example with the Exchange Online PowerShell command `Set-CASMailbox -Identity user@contoso.com -SmtpClientAuthenticationDisabled $false`.

Servers: SMTP `smtp.office365.com:587` (STARTTLS), IMAP `outlook.office365.com:993`, POP3 `outlook.office365.com:995`.

Gmail setup

1. In the Google Cloud Console create a project and configure the **OAuth consent screen** with the scope `https://mail.google.com/`. While the application is in testing mode, add the accounts which will sign in as test users.
2. In **Credentials** create an **OAuth client ID** of type **Desktop app**.
3. Copy the client ID to **ClientId** and the client secret to **ClientSecret**.
4. Use the authorization code flow with PKCE. Google does not allow the mail scope with the device code flow, **Start** raises an exception for that combination.

Servers: SMTP `smtp.gmail.com:587` (STARTTLS), IMAP `imap.gmail.com:993`, POP3 `pop.gmail.com:995`.

Flows

- `mofAuthorizationCodePKCE` (default): **Start** opens the browser on the sign in page of the provider and a local HTTP server, configured in **LocalServerOptions**, receives the redirect. The tokens are then requested and **OnTokensChanged** is fired.
- `mofDeviceCode`: for applications without a browser, services or consoles. **Start** fires **OnDeviceCode** with a short user code and a verification URI. Show them to the user, who signs in on any other device. When the sign in is complete **OnTokensChanged** is fired, and if the code expires first **OnError** is fired with `expired_token`. Microsoft 365 only.

With `mopCustom` the endpoints and the scope come from **CustomAuthURL**, **CustomTokenURL**, **CustomDeviceAuthorizationURL** and **CustomScope**.

Token persistence and OnTokensChanged

The user signs in only once. **OnTokensChanged** is fired every time new tokens are received, after **Start** and after every **Refresh**, with the access token, the refresh token and the UTC expiry time. Save the **RefreshToken** encrypted: it gives access to the mailbox. In the next session assign it to the **RefreshToken** property and call **Refresh**, which is synchronous, to get a new access token without user interaction. Access tokens are short lived (about one hour), refresh before **ExpiresAt**.
The token events can be fired in a secondary thread (the thread of the local HTTP server), synchronize any access to the user interface.

SASL XOAUTH2 and OAUTHBEARER

GetXOAuth2 returns the Base64 initial client response of the XOAUTH2 mechanism, `user={user}^Aauth=Bearer {token}^A^A`, supported by Microsoft 365 and Gmail. **GetOAuthBearer** returns the standard OAUTHBEARER response of RFC 7628, `n,a={user},^Ahost={host}^Aport={port}^Aauth=Bearer {token}^A^A`, supported by Gmail. The **Raw** variants return the same strings without Base64 encoding, for libraries which encode them. Send them as:

- SMTP: `AUTH XOAUTH2 {GetXOAuth2}`, success reply 235.
- IMAP: `AUTHENTICATE XOAUTH2 {GetXOAuth2}` or `AUTHENTICATE OAUTHBEARER {GetOAuthBearer}`.
- POP3: `AUTH XOAUTH2`, then the string when the server answers `+`.

The functions `sgcGetXOAuth2`, `sgcGetXOAuth2Raw`, `sgcGetOAuthBearer` and `sgcGetOAuthBearerRaw` of the same unit build the strings from an access token obtained in any other way.

Example: sending mail with Indy TIdSMTP

The strings are built from the current **AccessToken**. When there is no access token, **GetXOAuth2** and **GetOAuthBearer** raise "There is no access token, call Start or Refresh first.".

Reference

- Properties: 16 properties available on the component.
- Methods: 11 public methods.
- Events: 3 events you can handle.

Amazon AWS | SQS

What is Amazon SQS?

Amazon Simple Queue Service (SQS) is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications. SQS eliminates the complexity and overhead associated with managing and operating message oriented middleware, and empowers developers to focus on differentiating work. Using SQS, you can send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available.

Benefits

Eliminate administrative overhead

With SQS, there is no upfront cost, no need to acquire, install, and configure messaging software, and no time-consuming build-out and maintenance of supporting infrastructure.

Reliably deliver messages

SQS lets you decouple application components so that they run and fail independently, increasing the overall fault tolerance of the system.

Keep sensitive data secure

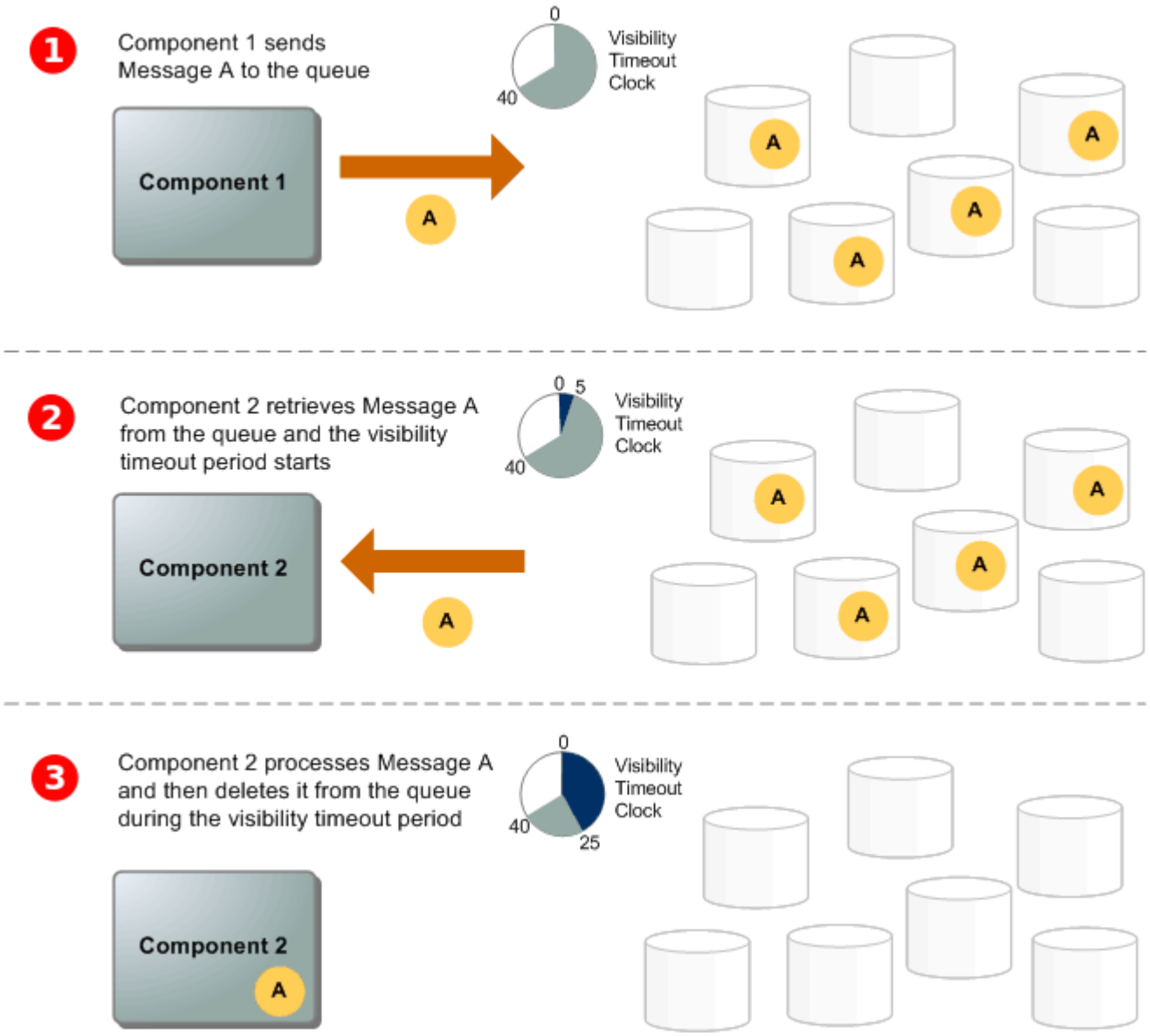
You can use Amazon SQS to exchange sensitive data between applications using server-side encryption (SSE) to encrypt each message body.

Scale elastically and cost-effectively

SQS scales elastically with your application so you don’t have to worry about capacity planning and pre-provisioning.

WorkFlow

The following scenario describes the lifecycle of an Amazon SQS message in a queue, from creation to deletion.



Getting Started with Amazon SQS

Before you begin, complete the steps in Setting Up Amazon SQS.

Step 1: Create a Queue

1. Sign in to the Amazon SQS console.
2. Choose Create New Queue.
3. On the Create New Queue page, ensure that you're in the correct region and then type the Queue Name.
4. Standard is selected by default. Choose FIFO.
5. To create your queue with the default parameters, choose Quick-Create Queue.

Your new queue is created and selected in the queue list.

Step 2: Send a Message

After you create your queue, you can send a message to it. The following example shows sending a message to an existing queue.

1. From the queue list, select the queue that you've created.
2. From Queue Actions, select Send a Message.
3. Your message is sent and the Send a Message to QueueName dialog box is displayed, showing the attributes of the sent message.

Step 3: Receive and Delete Your Message

After you send a message into a queue, you can consume it (retrieve it from the queue). When you request a message from a queue, you can't specify which message to get. Instead, you specify the maximum number of messages (up to 10) that you want to get.

Step 4: Delete Your Queue

If you don't use an Amazon SQS queue (and don't foresee using it in the near future), it is a best practice to delete it from Amazon SQS.

SQS Client

```
// TsgcHTTPAWS_SQS_Client is the component used for connect to Amazon SQS.
// Client connects using HTTPs protocol and authenticates using Access Key provided by Amazon.

// Before you attempt to connect to SQS service, you must set some data in AWSOptions property.

// Region: your endpoint region, example: us-east-1.
// AccessKey: access key provided by Amazon.
// SecretKey: secret key provided by Amazon.

// The following methods are supported by SQS client:

// AddPermission
// Adds a permission to a queue for a specific principal. This allows sharing access to the queue.

// ChangeMessageVisibility
// Changes the visibility timeout of a specified message in a queue to a new value. The default visibility
// The minimum is 0 seconds. The maximum is 12 hours.

// ChangeMessageVisibilityBatch

// Changes the visibility timeout of multiple messages. This is a batch version of ChangeMessageVisibility.
// The result of the action on each message is reported individually in the response. You can send up to 10

// CreateQueue
// Creates a new standard or FIFO queue. You can pass one or more attributes in the request.

  vURL = SQS->CreateQueue("sqs_queue");
  if (vURL != "")
  {
    DoLog("#CreateQueue: " + vURL);
  }

// DeleteMessage
// Deletes the specified message from the specified queue. To select the message to delete, use the Receipt

  if (SQS->DeleteMessage("sqs_queue", "...receipt handle goes here...") == true)
  {
    DoLog("#DeleteMessage: ok");
  }
  else
  {
    DoLog("#DeleteMessage: error");
  }

// DeleteMessageBatch
// Deletes up to ten messages from the specified queue. This is a batch version of DeleteMessage. The resul

// DeleteQueue
// Deletes the queue specified by the queue name, regardless of the queue's contents.

  if (SQS->DeleteQueue(txtQueueName->Text) == true)
  {
    DoLog("#Delete Queue: ok");
  }
  else
```

```
  {
    DoLog("#Delete Queue: error");
  }

// GetQueueAttributes
// Gets attributes for the specified queue.

  TsgcSQSAttributes *oAttributes = new TsgcSQSAttributes();
  try
  {
    if (SQS->GetQueueAttributes("sqs_queue", oAttributes) == true)
    {
      for (int i = 0; i < oAttributes->Count; i++)
      {
        DoLog("#Attribute: " + static_cast<TsgcSQSAttribute*>(oAttributes->Item[i])
          ->AttributeName + " " + static_cast<TsgcSQSAttribute*>(oAttributes->Item[i])
          ->AttributeValue);
      }
    }
    else
    {
      DoLog("#GetQueueAttributes: error");
    }
  }
  __finally
  {
      oAttributes->Free();
  }

// GetQueueUrl
// Returns the URL of an existing Amazon SQS queue.

// ListDeadLetterSourceQueues
// Returns a list of your queues that have the RedrivePolicy queue attribute configured with a dead-letter

// ListQueueTags
// List all cost allocation tags added to the specified Amazon SQS queue.

// PurgeQueue
// Deletes the messages in a queue specified by the QueueName parameter.

  if (SQS->PurgueQueue("sqs_queue") == true)
  {
    DoLog("#PurgueQueue: ok");
  }
  else
  {
    DoLog("#PurgueQueue: error");
  }

// ReceiveMessage
// Retrieves one or more messages (up to 10), from the specified queue.

  TsgcSQSReceiveMessageResponses *oResponses = new TsgcSQSReceiveMessageResponses();
  try
  {
    if (SQS->ReceiveMessage("sqs_test", oResponses) == true)
    {
      for (int i = 0; i < oResponses->Count; i++)
      {
        DoLog("#ReceiveMessage: " + static_cast<TsgcSQSReceiveMessageResponse*>(oResponses->Item[i])->Body)
        FReceiptHandle = static_cast<TsgcSQSReceiveMessageResponse*>(oResponses->Item[i])->ReceiptHandle;
      }
    }
  }
  __finally
  {
    oResponses->Free();
  }

// RemovePermission
// Revokes any permissions in the queue policy that matches the specified Label parameter.

// SendMessage
// Delivers a message to the specified queue.

  if (SQS->SendMessage("sqs_queue", "My First Message") == true)
  {
    DoLog("#SendMessage: ok");
  }
  else
  {
    DoLog("#SendMessage: error");
  }

// SendMessageBatch
```

```
// Delivers up to ten messages to the specified queue. This is a batch version of SendMessage.

// SetQueueAttributes
// Sets the value of one or more queue attributes. When you change a queue's attributes, the change can tak
// for most of the attributes to propagate throughout the Amazon SQS system.

  TsgcSQSAttributes *oAttributes = new TsgcSQSAttributes();
  try
  {
    oAttributes->AddSQSAttribute(sqsatVisibilityTimeout, "45");
    if (SQS->SetQueueAttributes("sqs_queue", oAttributes) == true)
    {
      DoLog("#SetQueueAttributes: ok");
    }
    else
    {
      DoLog("#SetQueueAttributes: error");
    }
  }
  __finally
  {
    oAttributes->Free();
  }

// TagQueue
// Add cost allocation tags to the specified Amazon SQS queue.

// UntagQueue
// Remove cost allocation tags from the specified Amazon SQS queue.
```

Events

OnSQSBeforeRequest

This event is called before sqs component does an HTTP request. You can get access to URL parameter and if Handled parameter is set to True, means component won't do an HTTP request.

OnSQSError

If there is any error when the component does a request, this event will be called with Error Code and Error Description.

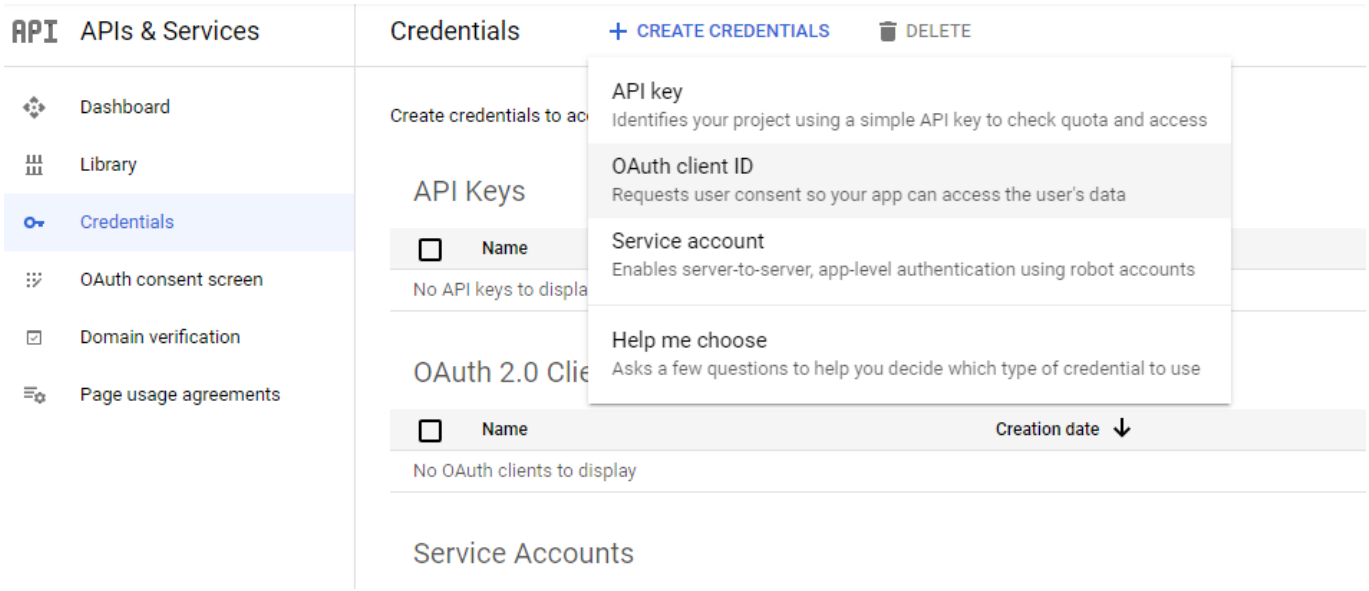
OnSQSResponse

This event is called after an HTTP request with raw response from server.

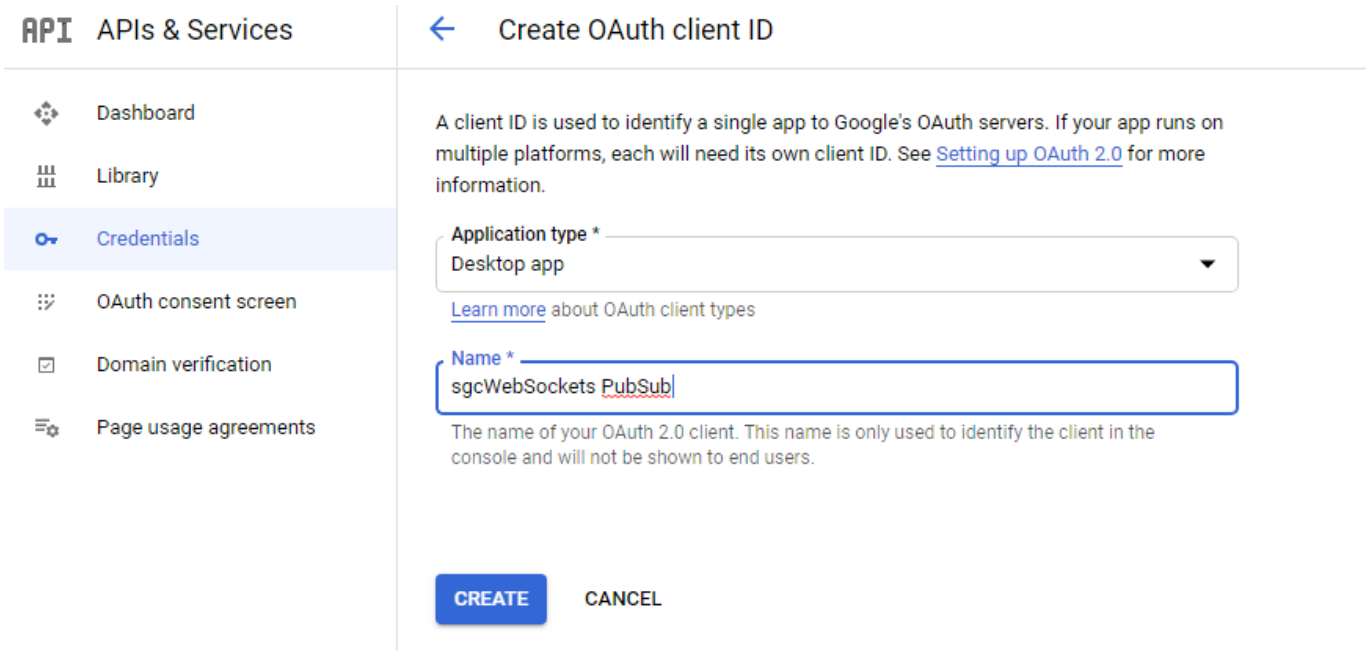
Google Cloud | Google OAuth2 Keys

In order to use the sgcWebSockets Google Cloud components and Authenticate using OAuth2, first you must obtain the OAuth2 Key from Google Cloud.
Find below the steps to get Google OAuth2 Keys and how to configure them in our PubSub sample application.

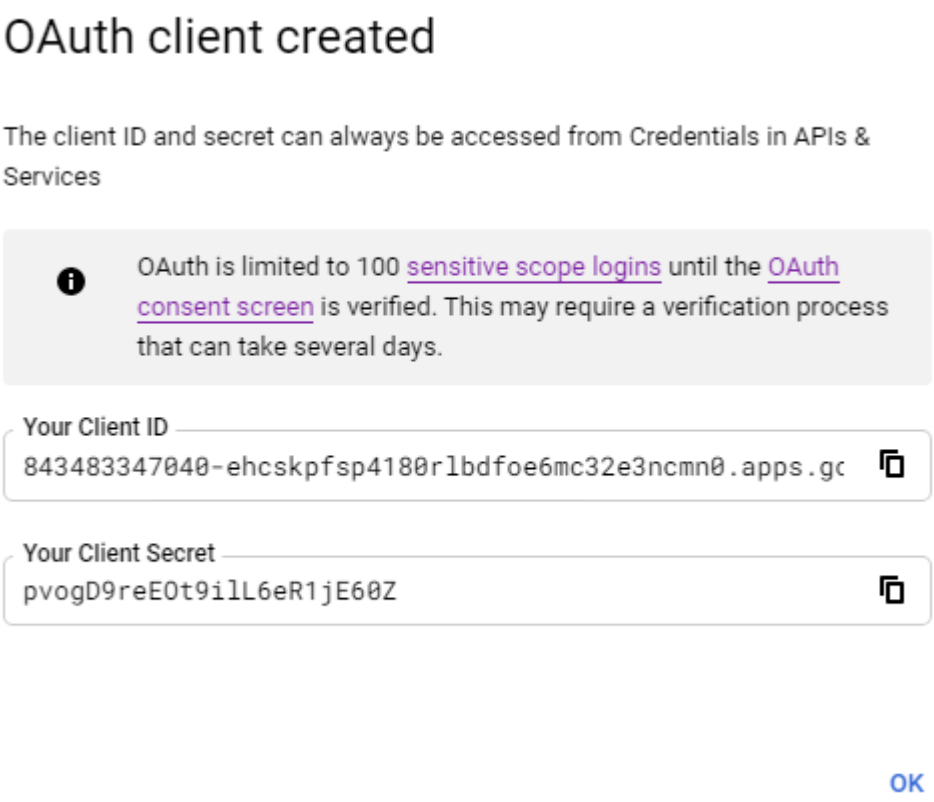
First **login** to your **Google Cloud Account** and use an existing project or create a new one.
After that, go to **Credentials** menu and press the button **CREATE CREDENTIALS**, select the option **OAuth Client ID**.



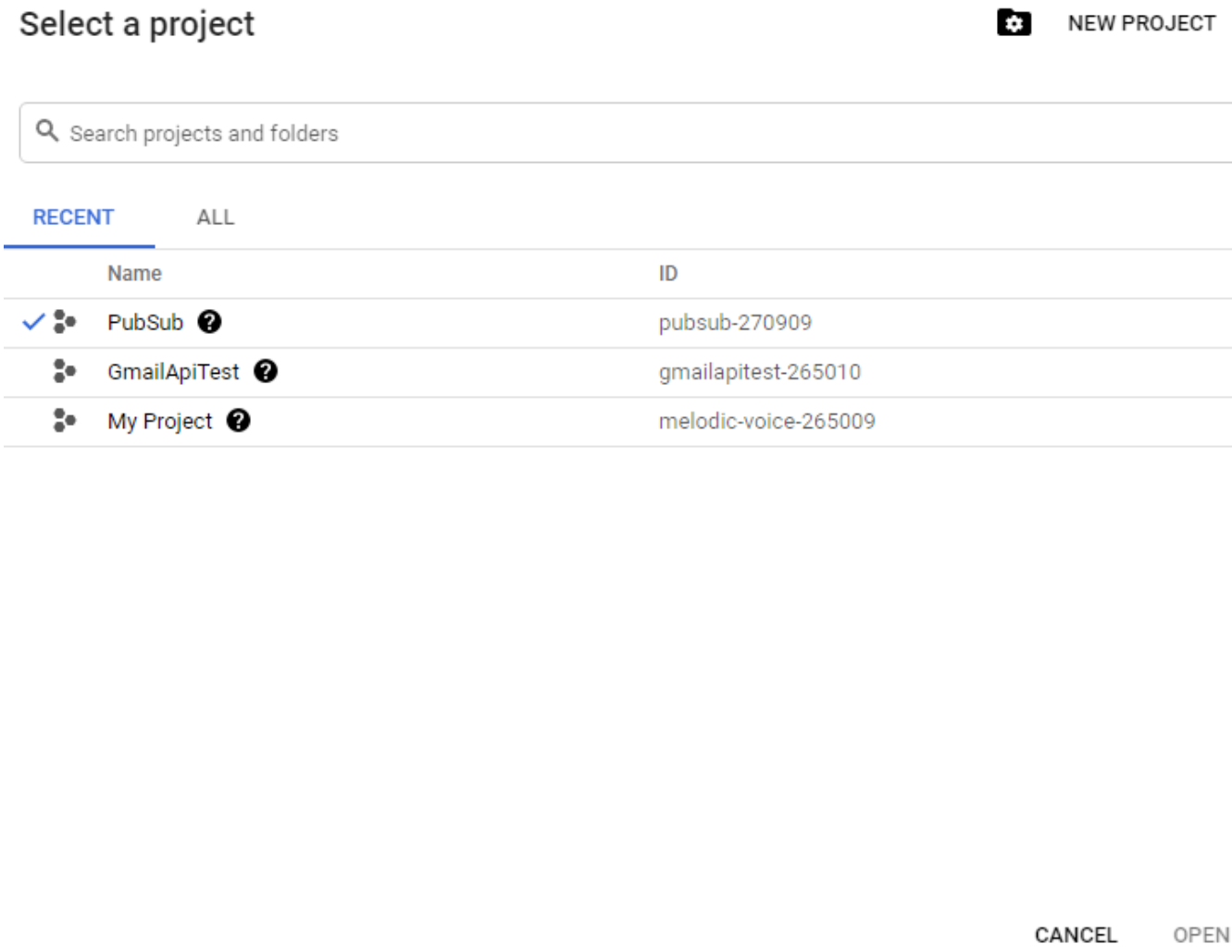
Select your application type and set a descriptive name.



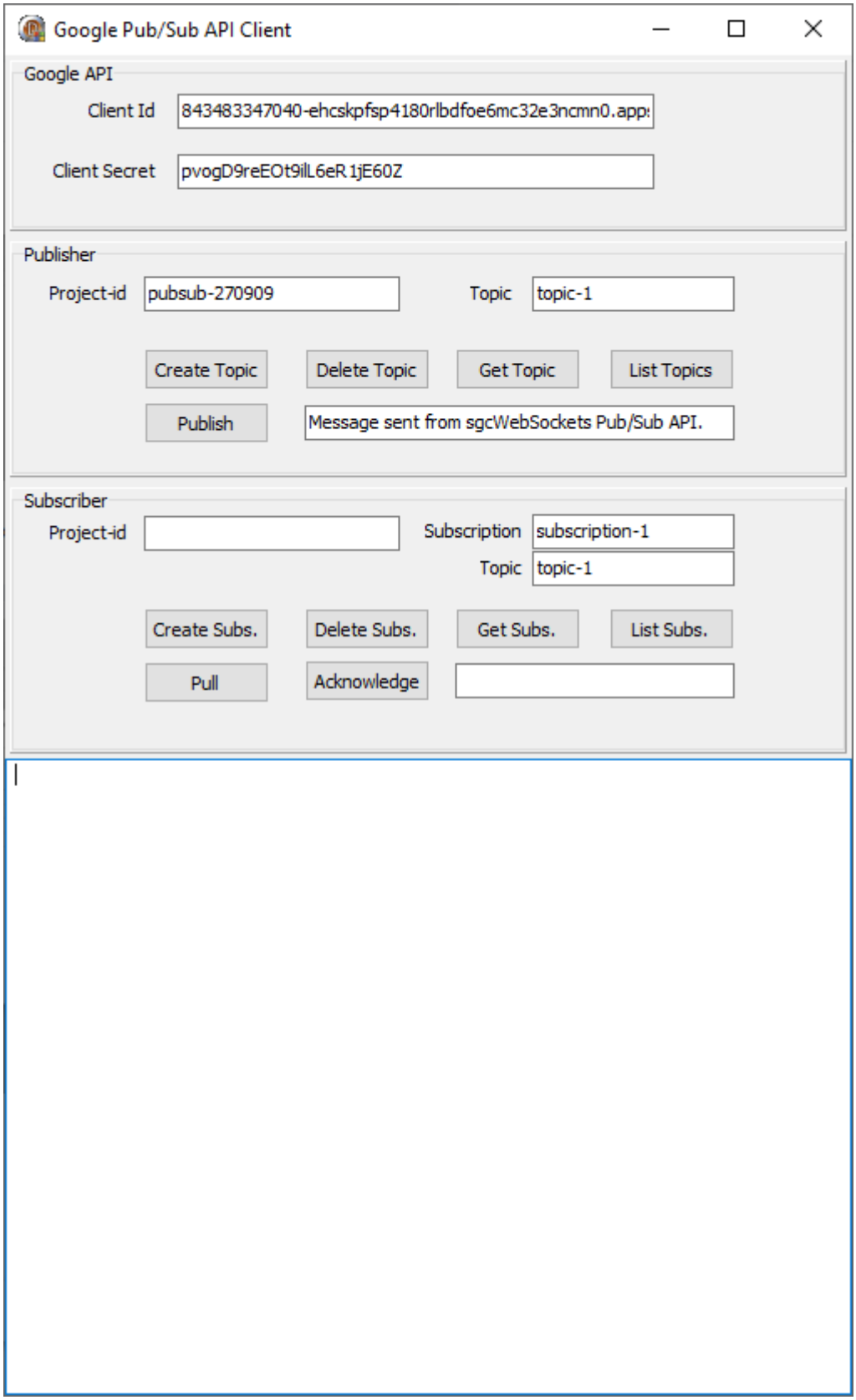
If successful, you will get your Client ID and Client Secret.



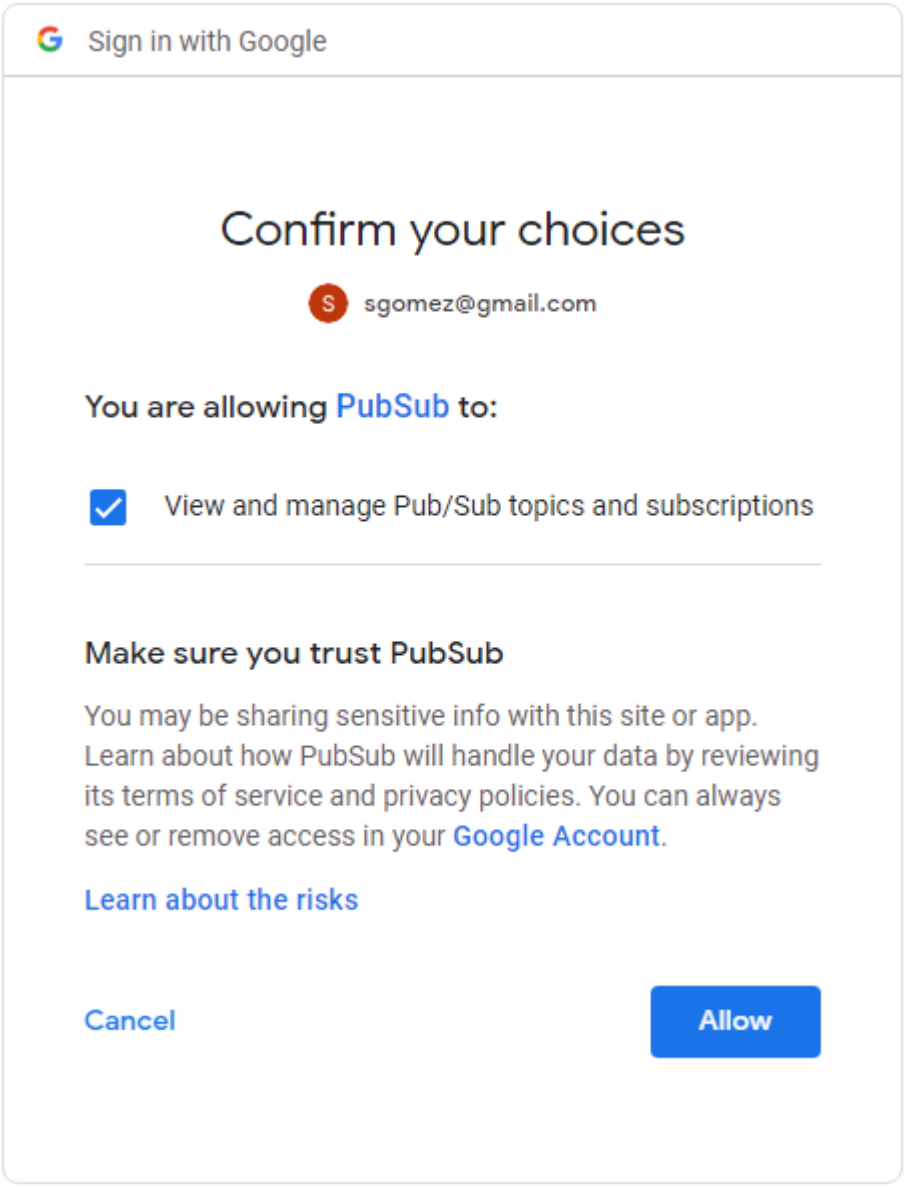
Don't share your OAuth2 data with anyone!
Now copy to the sgcWebSockets PubSub sample, and add the Project Id (NOT the project name)



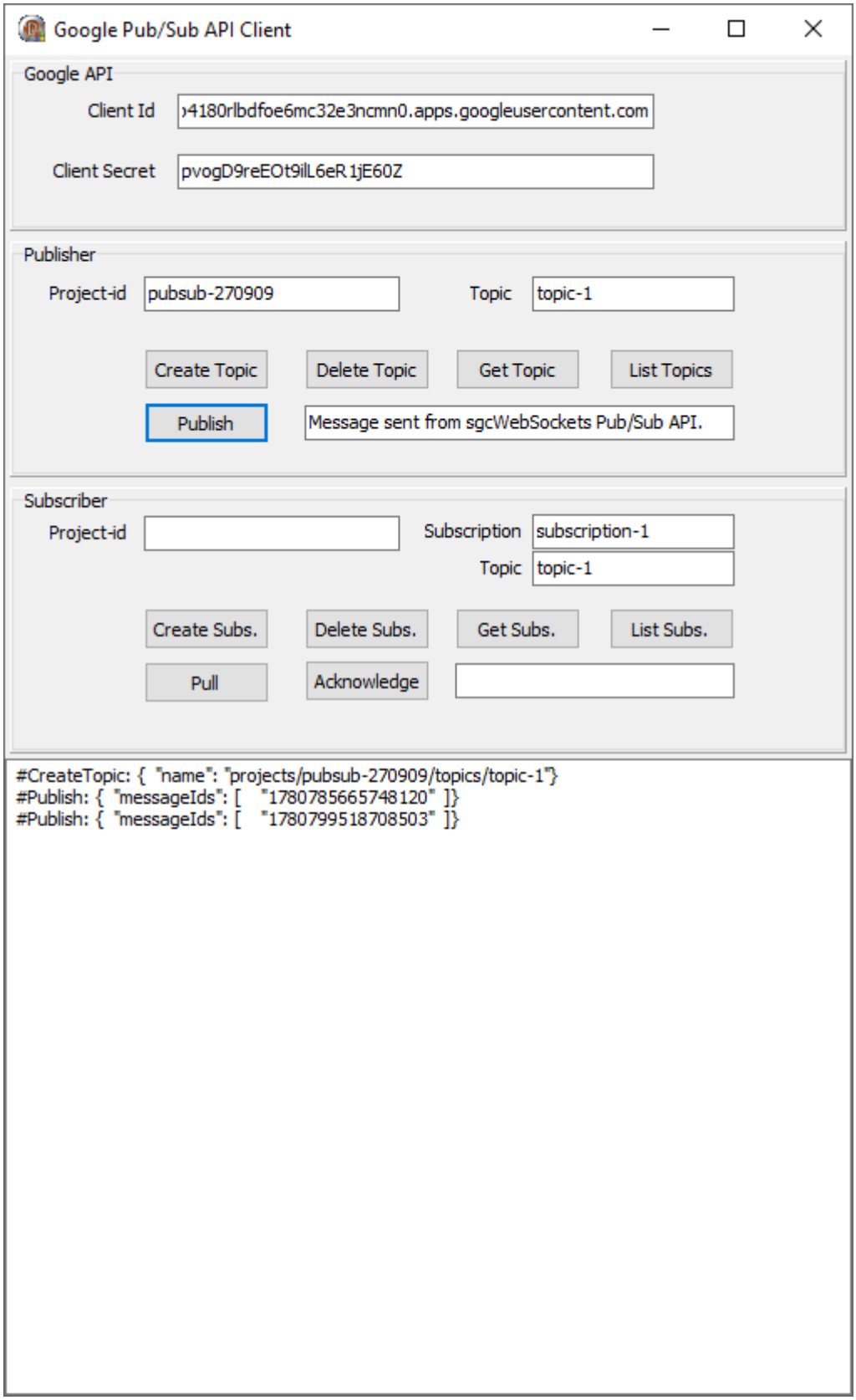
This is how it must be configured in the sgcWebSocket PubSub sample.



Then you can try to create a new topic, for example. The first time, you must authorize the OAuth2 connection, so a new web browser will be shown to request authorization to access your account with the OAuth2 credentials provided by Google.



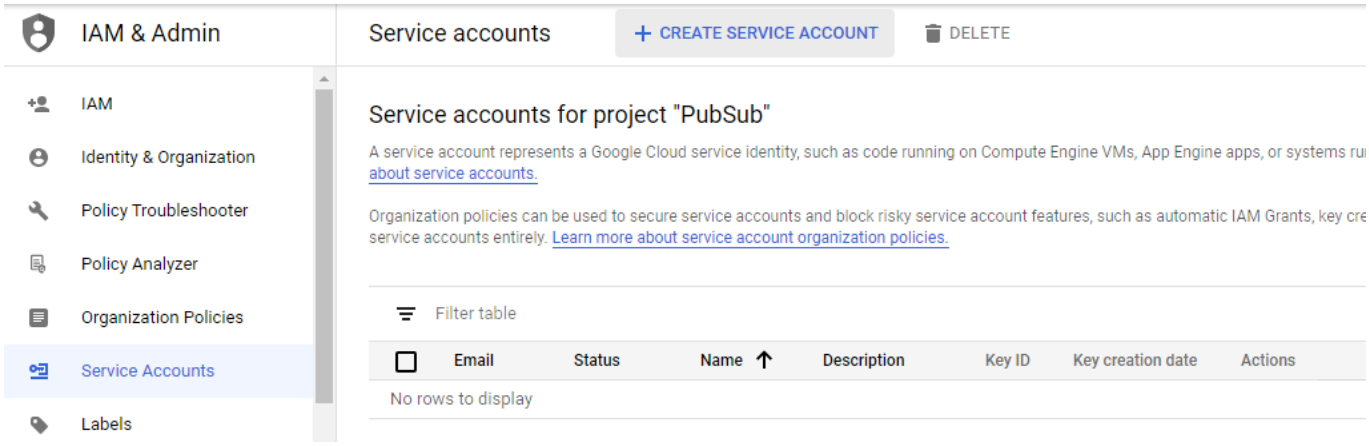
Allow the connection, and if successful, you can start working with this API.


Google Pub/Sub API Client
Google API
Client Id
›4180rlbdfoe6mc32e3ncmn0.apps.googleusercontent.com
Client Secret
pvogD9reEOt9ilL6eR1jE60Z
Publisher
Project-id
pubsub-270909
Topic
topic-1
Create Topic
Delete Topic
Get Topic
List Topics
Publish
Message sent from sgcWebSockets Pub/Sub API.
Subscriber
Project-id
Subscription
subscription-1
Topic
topic-1
Create Subs.
Delete Subs.
Get Subs.
List Subs.
Pull
Acknowledge
#CreateTopic: { "name": "projects/pubsub-270909/topics/topic-1"}
#Publish: { "messageIds": ["1780785665748120"]}
#Publish: { "messageIds": ["1780799518708503"]}

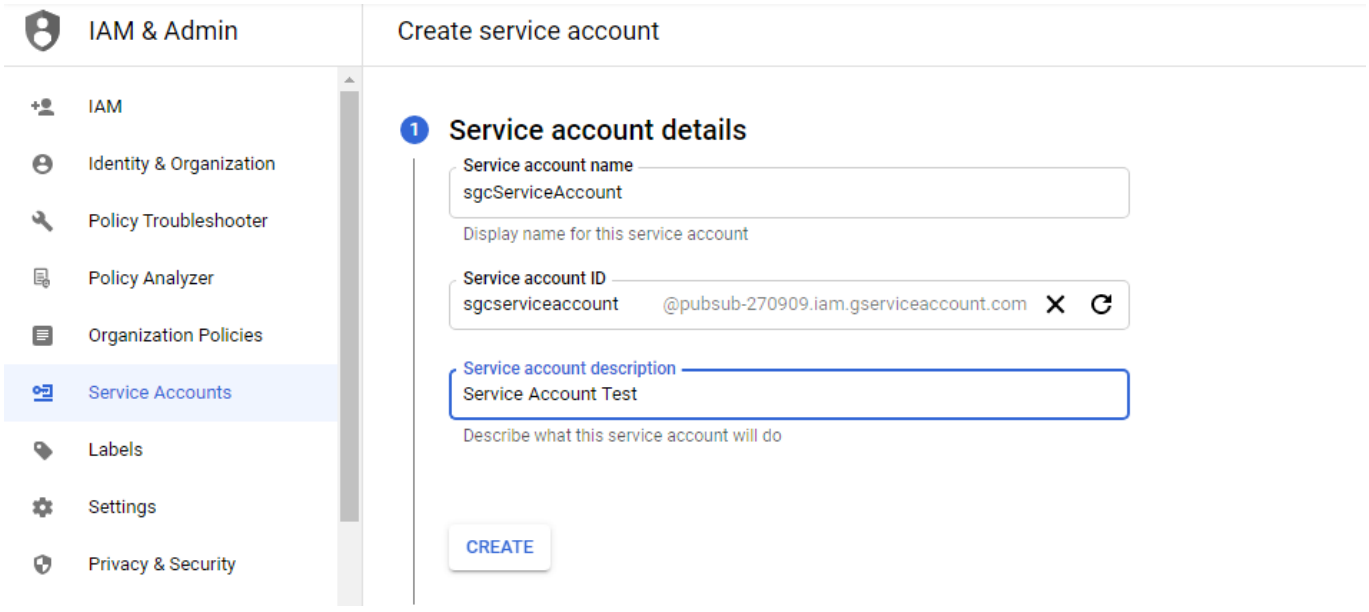
Google Cloud | Google Service Accounts

In order to use the sgcWebSockets Google Cloud components and authenticate using Service Accounts, first you must obtain the Private Key Certificate
from Google Cloud.
Find below the steps to get the Google Private Key Certificate and how to configure it in our PubSub sample application.

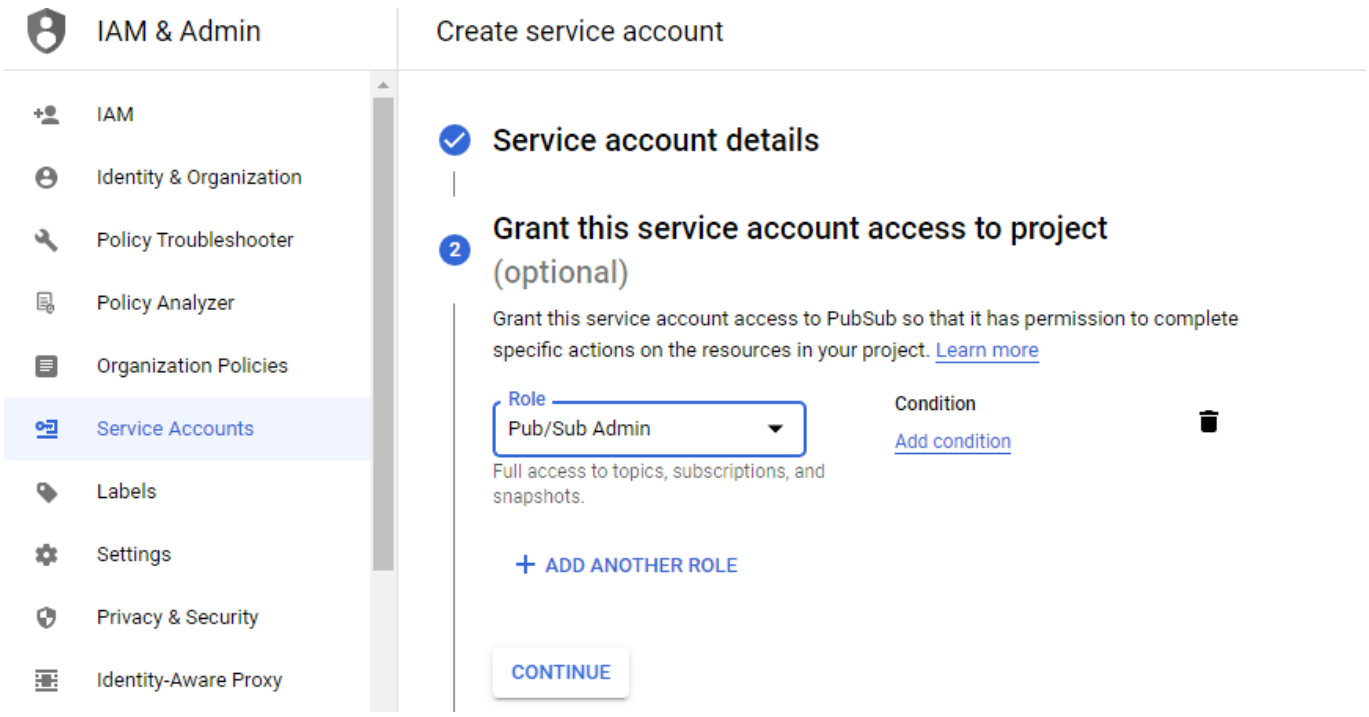
First **login** to your **Google Cloud Account** and use an existing project or create a new one.



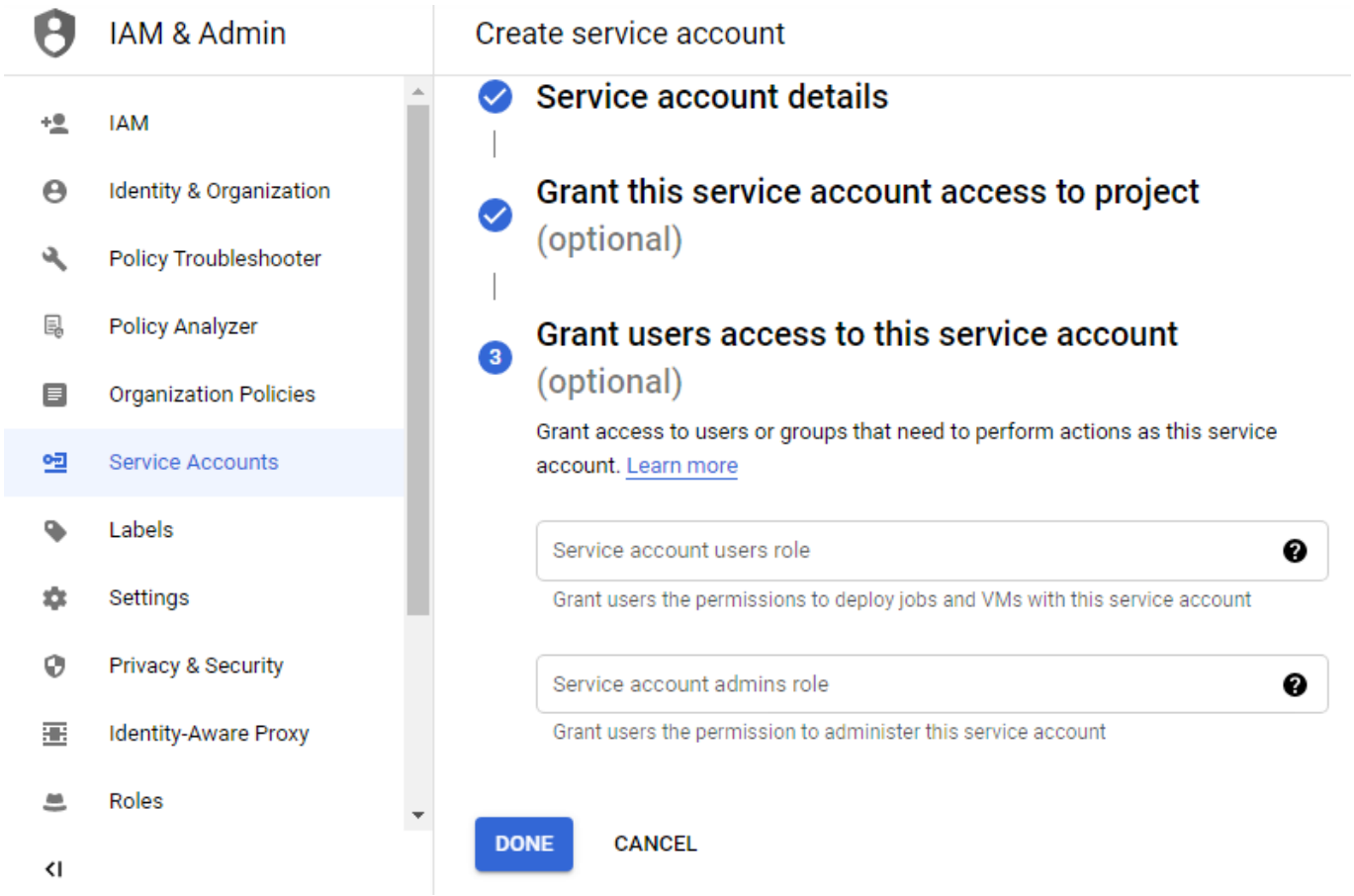
Select **CREATE SERVICE ACCOUNT** and a new page will be shown where you must set the service account name and description



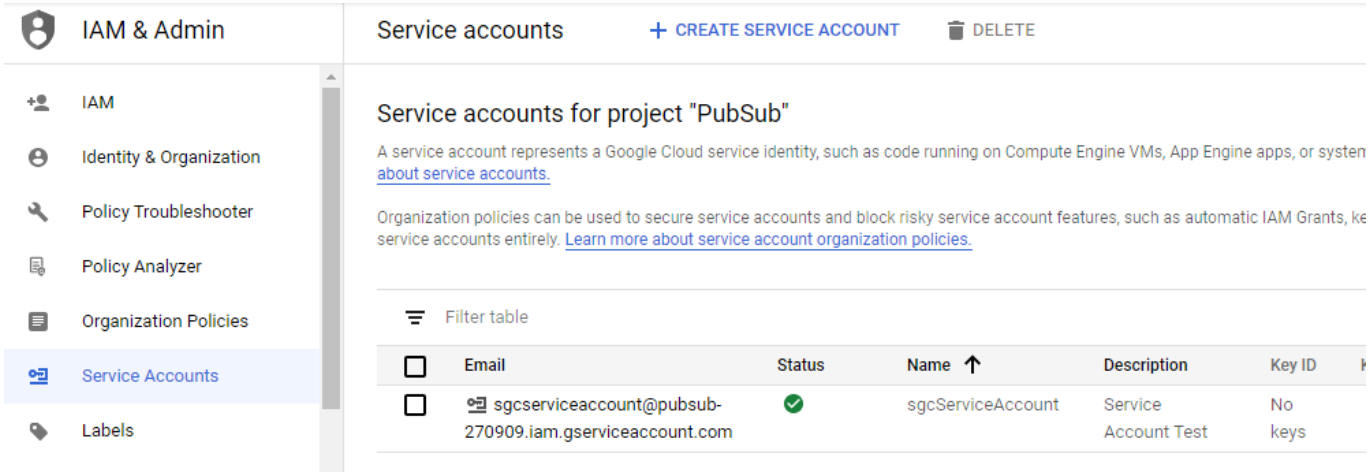
Then select at least one role. Here we select PubSub Admin to allow the client to publish and subscribe to topics, but you can select another role with fewer privileges.



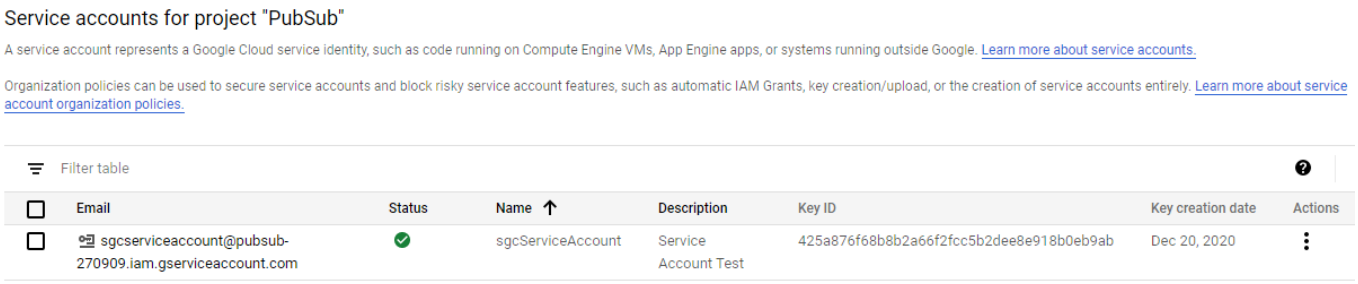
Press CONTINUE and finally you can grant access to other users



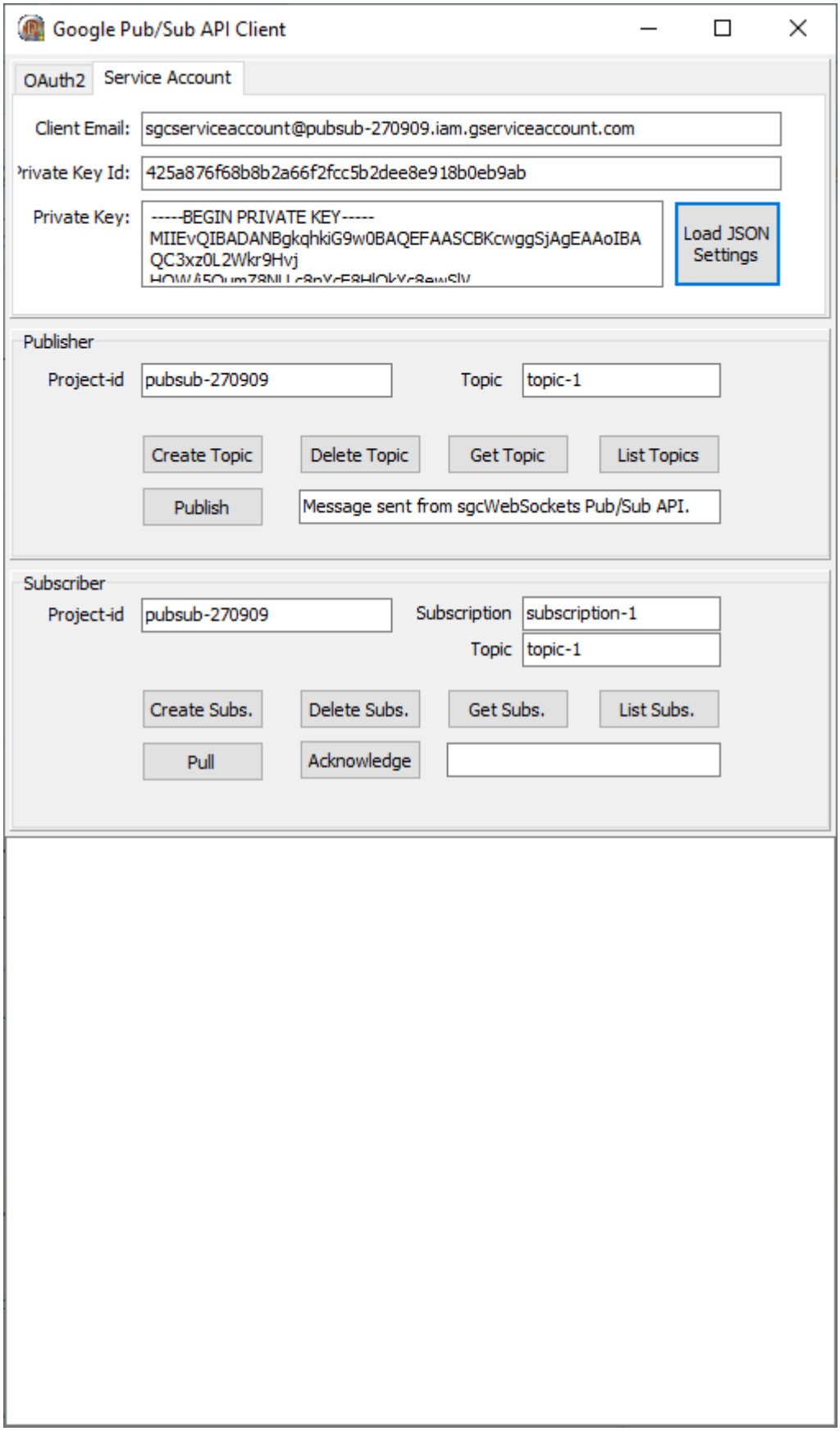
Press DONE when you finish, and a new record will be shown.



The next step is to create a new key, so select the Create Key option in the actions column. Select JSON to download the configuration in JSON format, and a new key will be created.



Finally, you only need to fill in the data provided by Google in the sgcWebSockets PubSub client. You can use **LoadSettingsFromFile** to load the configuration JSON file.


Google Pub/Sub API Client
OAuth2
Service Account
Client Email:
sgcserviceaccount@pubsub-270909.iam.gserviceaccount.com
rivate Key Id:
425a876f68b8b2a66f2fcc5b2dee8e918b0eb9ab
Private Key:
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBA
QC3xz0L2Wkr9Hvj
Load JSON Settings
Publisher
Project-id
pubsub-270909
Topic
topic-1
Create Topic
Delete Topic
Get Topic
List Topics
Publish
Message sent from sgcWebSockets Pub/Sub API.
Subscriber
Project-id
pubsub-270909
Subscription
subscription-1
Topic
topic-1
Create Subs.
Delete Subs.
Get Subs.
List Subs.
Pull
Acknowledge

Google Cloud | Pub/Sub

What is Google Cloud Pub/Sub?

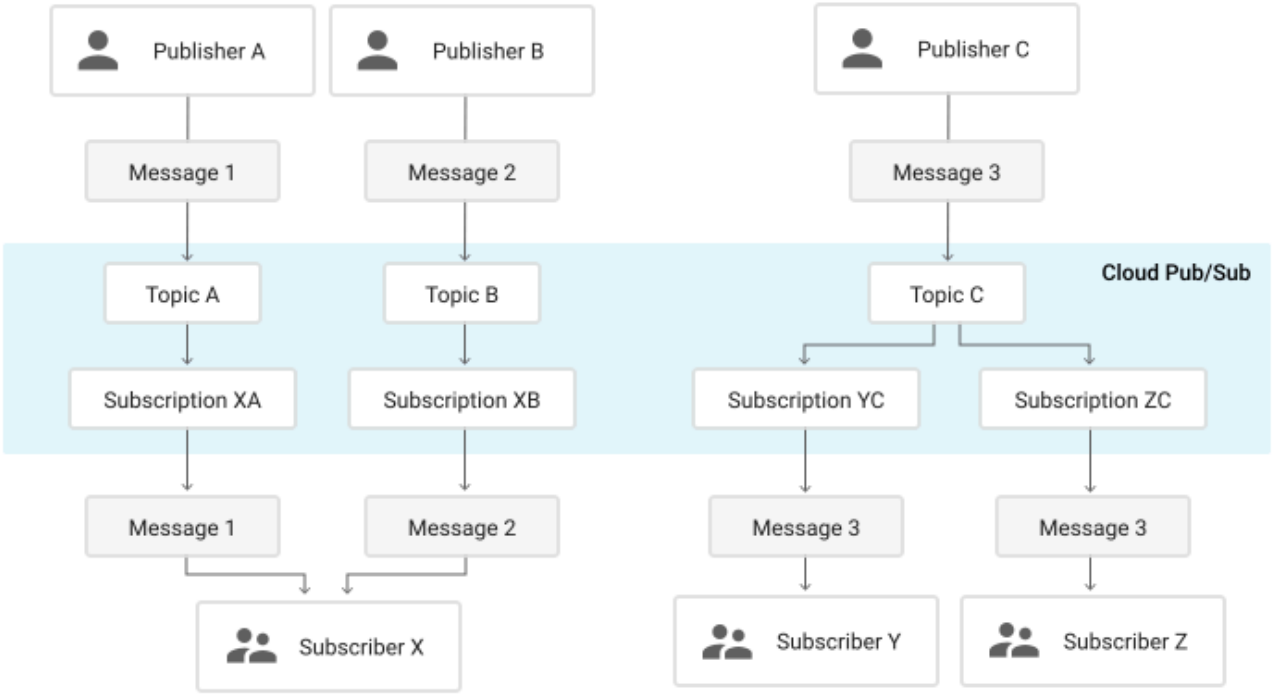
Pub/Sub brings the flexibility and reliability of enterprise message-oriented middleware to the cloud. At the same time, Pub/Sub is a scalable, durable event ingestion and delivery system that serves as a foundation for modern stream analytics pipelines. By providing many-to-many, asynchronous messaging that decouples senders and receivers, it allows for secure and highly available communication among independently written applications. Pub/Sub delivers low-latency, durable messaging that helps developers quickly integrate systems hosted on the Google Cloud Platform and externally.

Features

At-least-once delivery Synchronous, cross-zone message replication and per-message receipt tracking ensures at-least-once delivery at any scale.	**Open** Open APIs and client libraries in seven languages support cross-cloud and hybrid deployments.	**Exactly-once processing** Cloud Dataflow supports reliable, expressive, exactly-once processing of Cloud Pub/Sub streams.
Global by default Publish from anywhere in the world and consume from anywhere, with consistent latency. No replication necessary.	**No provisioning, auto-everything** Cloud Pub/Sub does not have shards or partitions. Just set your quota, publish, and consume.	**Compliance and security** Cloud Pub/Sub is a HIPAA-compliant service, offering fine-grained access controls and end-to-end encryption.
Integrated Take advantage of integrations with multiple services, such as Cloud Storage and Gmail update events and Cloud Functions for serverless event-driven computing.	**Seek and replay** Rewind your backlog to any point in time or a snapshot, giving the ability to reprocess the messages. Fast forward to discard outdated data.	

Publisher-subscriber relationships

A publisher application creates and sends messages to a topic. Subscriber applications create a subscription to a topic to receive messages from it. Communication can be one-to-many (fan-out), many-to-one (fan-in), and many-to-many.



Common use cases

- **Balancing workloads in network clusters.** For example, a large queue of tasks can be efficiently distributed among multiple workers, such as Google Compute Engine instances.
- **Implementing asynchronous workflows.** For example, an order processing application can place an order on a topic, from which it can be processed by one or more workers.
- **Distributing event notifications.** For example, a service that accepts user signups can send notifications whenever a new user registers, and downstream services can subscribe to receive notifications of the event.
- **Refreshing distributed caches.** For example, an application can publish invalidation events to update the IDs of objects that have changed.
- **Logging to multiple systems.** For example, a Google Compute Engine instance can write logs to the monitoring system, to a database for later querying, and so on.
- **Data streaming from various processes or devices.** For example, a residential sensor can stream data to backend servers hosted in the cloud.
- **Reliability improvement.** For example, a single-zone Compute Engine service can operate in additional zones by subscribing to a common topic, to recover from failures in a zone or region.

Authorization

Google Pub/Sub component client can login to Google Servers using the following methods:

- **gcaOAuth2:** OAuth2 protocol
- **gcaJWT:** JWT tokens.

OAuth2

The login is done using a web browser where the user logs in with their own account and authorizes the PubSub requests.

- **GoogleCloudOptions.OAuth2.ClientId:** is the ClientID provided by Google to Authenticate through OAuth2 protocol.
- **GoogleCloudOptions.OAuth2.ClientSecret:** is the Client Secret string provided by Google to Authenticate through OAuth2 protocol.

- **GoogleCloudOptions.OAuth2.Scope:** is the scope of OAuth2, usually there is no need to modify the default value unless you need to get more access than default.
- **GoogleCloudOptions.OAuth2.LocalIP:** the OAuth2 protocol requires a local server listening for the response from the authentication server. This is the IP or DNS name. By default, it is 127.0.0.1.
- **GoogleCloudOptions.OAuth2.LocalPort:** Local server listening port.
- **GoogleCloudOptions.OAuth2.RedirectURL:** if you need to set a redirect URL different from LocalPort + LocalIP, you can set it in this property (example: http://127.0.0.1:8080/oauth2).

Service Accounts

The login is done by signing the requests using a private key provided by Google. This method is recommended for automated services or applications without user interaction.

- **GoogleCloudOptions.JWT.ClientEmail:** the client email name provided when creating the new service account. "client_email" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKeyId:** the Private Key ID provided by Google. "private_key_id" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKey:** the Private Key certificate provided by Google. "private_key" node in the JSON configuration file.

TLS Options

TLSOptions centralizes the configuration of the secure channel used to communicate with Google Cloud Pub/Sub.

- **TLSOptions.IOHandler:** pick the TLS implementation that fits your deployment (OpenSSL, SChannel...).
- **TLSOptions.Version:** restrict the negotiation to a given TLS version when your infrastructure requires it.
- **TLSOptions.VerifyCertificate:** control certificate validation, enabling stricter security policies.
- **TLSOptions.OpenSSL_Options.LibPath:** set the directory that contains the OpenSSL binaries shipped with your application.
- **TLSOptions.SChannel_Options.UseLegacyCredentials:** activates Windows legacy credential handling when SChannel requires it.

Example:

```
TsgcHTTPGoogleCloud_PubSub_Client *oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client(NULL);
oPubSub->TLSOptions->IOHandler = iohOpenSSL;
oPubSub->TLSOptions->Version = tls1_3;
oPubSub->TLSOptions->VerifyCertificate = true;
oPubSub->TLSOptions->OpenSSL_Options->LibPath = oslpDefaultFolder;
```

When a new service account is created, you can download a JSON file with all configurations. This file can be processed by the PubSub component, just call the method **LoadSettingsFromFile** and pass the JSON filename as argument.

Most common uses

- **Configuration**
  - **Google OAuth2 Keys**
  - **Service Accounts**

Google Pub/Sub Client

OAuth2

In order to work with Google Pub/Sub API, sgcWebSockets Pub/Sub component uses OAuth2 as default authentication, so first you must set your **ClientId** and **ClientSecret** from your google account.

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->Authorization = gcaOAuth2;
```

```
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
```

Service Accounts

Service Accounts require building a JWT and passing it as an authorization token.

```
TsgcHTTPGoogleCloud_PubSub_Client *oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client(this);
oPubSub->GoogleCloudOptions->Authorization = gcaJWT;
oPubSub->GoogleCloudOptions->JWT->ClientEmail = "...google email...";
oPubSub->GoogleCloudOptions->JWT->PrivateKeyId = "...private key id...";
oPubSub->GoogleCloudOptions->JWT->PrivateKey->Lines->Text = "...private key certificate...";
```

This is required in order to get an Authorization Token Key from Google which will be used for all Rest API calls.

All methods return a response, which may be successful or return an error.

Projects.Snapshots

Method	Parameters	Description	Example
CreateSnapshot	project, snapshot, subscription	Creates a snapshot from the requested subscription. Snapshots are used in subscriptions.seek operations, which allow you to manage message acknowledgments in bulk. That is, you can set the acknowledgment state of messages in an existing subscription to the state captured by a snapshot.	`CreateSnapshot('pubsub-270909', 'snapshot-1', 'subscription-1')`
DeleteSnapshot	project, snapshot	Removes an existing snapshot	`DeleteSnapshot('pubsub-270909', 'snapshot-1')`
ListSnapshots	project	Lists the existing snapshots	`ListSnapshots('pubsub-270909')`

Projects.Subscriptions

Method	Parameters	Description	Example
AcknowledgeSubscription			
CreateSubscription	project, subscription, topic	Creates a subscription to a given topic. If the subscription already exists, returns ALREADY_EXISTS. If the corresponding topic doesn't exist, returns NOT_FOUND.	`CreateSubscription('pubsub-270909', 'subscription-1', 'topic-1')`
DeleteSubscripton	project, subscription	Deletes an existing subscription. All messages retained in the subscription are immediately dropped.	`DeleteSubscription('pubsub-270909', 'subscription-1')`
GetSubscription	project, subscription	Gets the configuration details of a subscription.	`GetSubscription('pubsub-270909', 'subscription-1')`
ListSubscriptions	project	Lists matching subscriptions.	`ListSubscriptions('pubsub-270909', 'subscription-1')`
ModifyAckDeadlineSubscription	project, subscription, AckIds	Modifies the ack deadline for a specific message. This method is useful to indicate that more time is needed to process a message by the subscriber, or to make the message available for redelivery if the processing was interrupted. Note that this does not modify the subscription-level ackDeadlineSeconds used for	

		subsequent messages.	
Modify-Push-Config-Subscription	project, subscription	Modifies the PushConfig for a specified subscription.This may be used to change a push subscription to a pull one (signified by an empty PushConfig) or vice versa, or change the endpoint URL and other attributes of a push subscription. Messages will accumulate for delivery continuously through the call regardless of changes to the PushConfig.	
Pull	project, subscription	Pulls messages from the server. The server may return UNAVAILABLE if there are too many concurrent pull requests pending for the given subscription.	`pull('pubsub-270909', 'subscription-1')`
Seek	project, subscription, timeUTC, snapshot	Seeks an existing subscription to a point in time or to a given snapshot, whichever is provided in the request. Snapshots are used in subscriptions.seek operations, which allow you to manage message acknowledgments in bulk. That is, you can set the acknowledgment state of messages in an existing subscription to the state captured by a snapshot. Note	

		that both the subscription and the snapshot must be on the same topic.	

Projects.Topics

Method	Parameters	Description	Example
CreateTopic	project, topic	Creates the given topic with the given name	`CreateTopic('pubsub-270909', 'topic-1')`
DeleteTopic	project, topic	Deletes the topic with the given name. Returns NOT_FOUND if the topic does not exist. After a topic is deleted, a new topic may be created with the same name; this is an entirely new topic with none of the old configuration or subscriptions.	`DeleteTopic('pubsub-270909', 'topic-1')`
GetTopic	project, topic	Gets the configuration of a topic.	`GetTopic('pubsub-270909', 'topic-1')`
ListTopics	project	Lists matching topics.	`ListTopics('pubsub-270909')`
Publish	project, topic, message	Adds one or more messages to the topic. Returns NOT_FOUND if the topic does not exist.	`Publish('pubsub-270909', 'topic-1', 'My First PubSub Message.')`

Projects.Topics.Subscriptions

Method	Parameters	Description	Example
ListTopicSubscriptions	project, topic	Lists the names of the subscrip-	`ListTopicSubscriptions('pubsub-270909', 'topic-1')`

		tions on this top-ic.	

Most common methods

Find below the most common methods used with the Google Cloud Pub/Sub API.

How to create a new Topic

Create a new topic for project with id: pubsub-270909 and topic name topic-1.

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
oPubSub->CreateTopic("pubsub-270909", "topic-1");
```

Response from Server

```
{
  "name": "projects/pubsub-270909/topics/topic-1"
}
```

Publish a message

Publish a new message in the newly created topic.

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
oPubSub->Publish("pubsub-270909", "topic-1", "My First Message from sgcWebSockets."));
```

Response from Server

```
{
  "messageIds": [
    "1050732082561505"
  ]
}
```

Publish a Message with Attributes

```
TsgcHTTPGoogleCloud_PubSub_Client *oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
oAttributes = new TStringList();
try
{
  oAttributes->CommaText = "origin=gcloud-sample,username=gcp";
  oPubSub->Publish("pubsub-270909", "topic-1", "My First Message from sgcWebSockets.", oAttributes, "username"));
}
__finally
{
  oAttributes->Free();
}
```

How to Create a new Subscription

Create a new subscription for project with id: pubsub-270909, with subscription name subscription-1 and topic-1

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
oPubSub->CreateSubscription("pubsub-270909", "subscription-1", "topic-1");
```

Response from Server

```
{
  "name": "projects/pubsub-270909/subscriptions/subscription-1",
  "topic": "projects/pubsub-270909/topics/topic-1",
  "pushConfig": {},
  "ackDeadlineSeconds": 10,
  "messageRetentionDuration": "604800s",
  "expirationPolicy": {
    "ttl": "2678400s"
  }
}
```

How to Read messages from a Subscription

Read messages from previous subscription created.

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
oPubSub->pubsub->Pull("pubsub-270909", "subscription-1");
```

Response from Server

```
{
  "receivedMessages": [
    {
      "ackId": "PjA-RVNEUAYWLF1GSFE3GQhoUQ5PXiM_NSAoRREFC08CKF15MEorQVh0Dj4N",
      "message": {
        "data": "TXkgRmlyc3QgTWVzc2FnZSBmcm9tIHNnY1dlYlNvY2tldHMu",
        "messageId": "1050732082561505",
        "publishTime": "2020-03-14T15:25:31.505Z"
      }
    }
  ]
}
```

Message is received encoded in Base64, so you must decode first to read contents.

```
sgcBase_Helpers->DecodeBase64("TXkgRmlyc3QgTWVzc2FnZSBmcm9tIHNnY1dlYlNvY2tldHMu=");
```

Google Cloud | Calendar

The Google Calendar API lets you integrate your app with Google Calendar, creating new ways for you to engage your users. The Calendar API lets you display, create and modify calendar events as well as work with many other calendar-related objects, such as calendars or access controls.

API Resources

Google Calendar uses the following resources:

- **Event:** An event on a calendar containing information such as the title, start and end times, and attendees. Events can be either single events or recurring events. An event is represented by an Event resource. The Events collection for a given calendar contains all event resources for that calendar.
- **Calendar:** A calendar is a collection of events. Each calendar has associated metadata, such as calendar description or default calendar time zone. The metadata for a single calendar is represented by a Calendar resource. The Calendars collection contains Calendar resources for all existing calendars.
- **CalendarList:** A list of all calendars on a user's calendar list in the Calendar UI. The metadata for a single calendar that appears on the calendar list is represented by a CalendarListEntry resource. This metadata includes user-specific properties of the calendar, such as its color or notifications for new events. The CalendarList collection contains all CalendarListEntry resources for a given user. For a further explanation of the difference between the Calendars and CalendarList collections, see Calendar and Calendar List
- **Setting:** A user preference from the Calendar UI, such as the user's time zone. A single user preference is represented by a Setting Resource. The Settings collection contains all Setting resources for a given user.
- **ACL:** An access control rule granting a user (or a group of users) a specified level of access to a calendar. A single access control rule is represented by an ACL resource. The ACL collection for a given calendar contains all ACL resources that grant access to that calendar.
- **Color:** A color presented in the Calendar UI. The Colors resource represents the set of all colors available in the Calendar UI, in two groups: colors available for events and colors available for calendars.
- **Free/busy:** A time when a calendar has events scheduled is considered "busy", a time when a calendar has no events is considered "free". The Freebusy resource allows querying for the set of busy times for a given calendar or set of calendars.

Main Features

- Fully Featured Google Calendar Client API V3.
- All Methods supported by API can be called using client API.
- Client requests using HTTP/2 protocol (*only Enterprise Edition).
- Automatic Handling of partial responses using PageNextToken.
- Easy access to Calendar and Event data properties.
- Authentication methods:
    - OAuth2: requires user interaction.
    - Service Accounts (requires Domain-Wide Delegation): for windows services, daemons...

Configuration

Google Calendar component client has the following properties:

OAuth2

- **GoogleCloudOptions.OAuth2.ClientId:** is the ClientID provided by Google to Authenticate through OAuth2 protocol.
- **GoogleCloudOptions.OAuth2.ClientSecret:** is the Client Secret string provided by Google to Authenticate through OAuth2 protocol.
- **GoogleCloudOptions.OAuth2.Scope:** is the scope of OAuth2, usually there is no need to modify the default value unless you need to get more access than default.
- **GoogleCloudOptions.OAuth2.LocalIP:** the OAuth2 protocol requires a local server listening for the response from the authentication server. This is the IP or DNS name. By default, it is 127.0.0.1.

- **GoogleCloudOptions.OAuth2.LocalPort:** Local server listening port.
- **GoogleCloudOptions.OAuth2.RedirectURL:** if you need to set a redirect URL different from LocalPort + LocalIP, you can set it in this property (example: http://127.0.0.1:8080/oauth2).

You can modify the Scopes of your client API using Scopes property, just select which scopes are supported by your client.

JWT

The login is done signing the requests using a private key provided by google, this method is recommended for automated services or applications without user interaction. Requires configure the Service Account with Domain-Wide Delegation.

- **GoogleCloudOptions.JWT.ClientEmail:** the client email name provided when creating the new service account. "client_email" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKeyId:** the Private Key ID provided by Google. "private_key_id" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKey:** the Private Key certificate provided by Google. "private_key" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.Subject:** is the workspace email account linked to the service account using Domain-Wide Delegation.

TLS Options

Use the **TLSOptions** property to customize the secure connection established with Google servers.

- **TLSOptions.IOHandler:** selects the TLS stack (OpenSSL, SChannel...).
- **TLSOptions.Version:** forces a specific TLS protocol version (for example tls1_2 or tls1_3).
- **TLSOptions.VerifyCertificate:** enables or disables server certificate validation.
- **TLSOptions.OpenSSL_Options.LibPath:** points to the folder that contains the OpenSSL libraries deployed with the application.
- **TLSOptions.SChannel_Options.UseLegacyCredentials:** enables the legacy credential flow required by some Windows versions when using SChannel.

The following snippets show how to configure the TLS options:

```
TsgcHTTPGoogleCloud_Calendar_Client *oCalendar = new TsgcHTTPGoogleCloud_Calendar_Client(NULL);
oCalendar->TLSOptions->IOHandler = iohOpenSSL;
oCalendar->TLSOptions->Version = tls1_3;
oCalendar->TLSOptions->VerifyCertificate = true;
oCalendar->TLSOptions->OpenSSL_Options->LibPath = oslpDefaultFolder;
```

Most common uses

- **Configuration**
  - Google Calendar Refresh Token
  - Google Calendar Service Account
- **Synchronization**
  - Google Calendar Sync Calendars
  - Google Calendar Sync Events

Synchronize Calendars

TsgcHTTPGoogleCloud_Calendar_Client component allows you to synchronize the calendars using direct Google API calls or using our easy Calendars methods to synchronize the calendars.

Method	Parameters	Description

NewCalendar	**aSummary:** the title of the calendar.	Creates a new Calendar
DeleteCalendar	**aId:** identifier of the calendar.	Deletes an existing Calendar.
UpdateCalendar	**aResource:** object with the calendar data.	Updates an existing Calendar.
LoadCalendars		Loads all calendars and Calendars property is filled with this data.
LoadCalendarsChanged	**aSyncToken:** last token used to update your calendar.	Loads all changes in your calendars from Token set.

Calendar Client has a property called **Calendars**, where you can access the calendar data after calling any of the previous methods. This property is synchronized automatically.

Synchronize Events

TsgcHTTPGoogleCloud_Calendar_Client component allows you to synchronize the events using direct Google API calls or using our easy Event methods to synchronize the Events.

Method	Parameters	Description
NewEvent	**aCalendarId:** id of the calendar. **aResource:** object with the event data.	Creates a new Event.
DeleteEvent	**aCalendarId:** id of the calendar. **aId:** identifier of the event.	Deletes an existing Event.
UpdateEvent	**aCalendarId:** id of the calendar. **aResource:** object with the event data.	Updates an existing Event.
LoadEvents	**aCalendarId:** id of the calendar.	Loads all events of the calendar.
LoadEventsChanged	**aCalendarId:** id of the calendar. **aSyncToken:** last token used to update your calendar.	Loads all events of the calendar from Token set.

You can access event data using the **Calendars** property. Select any of the existing calendars from the list and access its **Events** property.

Google Calendar API Calls

Method	Description
ACL_Delete	Deletes an access control rule.
ACL_Get	Returns an access control rule.
ACL_Insert	Creates an access control rule.
ACL_List	Returns the rules in the access control list for the calendar.
ACL_Patch	Updates an access control rule. This method supports patch semantics.
ACL_Update	Updates an access control rule.
ACL_Watch	Watch for changes to ACL resources.

Method	Description
CalendarList_Delete	Removes a calendar from the user's calendar list.
CalendarList_Get	Returns a calendar from the user's calendar list.
CalendarList_Insert	Inserts an existing calendar into the user's calendar list.
CalendarList_List	Returns the calendars on the user's calendar list.
CalendarList_Patch	Updates an existing calendar on the user's calendar list. This method supports patch semantics.
CalendarList_Update	Updates an existing calendar on the user's calendar list.
CalendarList_Watch	Watch for changes to CalendarList resources.

Method	Description
Calendar_Clear	Clears a primary calendar. This operation deletes all events associated with the primary calendar of an account.

Calendar_Delete	Deletes a secondary calendar. Use calendars.clear for clearing all events on primary calendars.
Calendar_Get	Returns metadata for a calendar.
Calendar_Insert	Creates a secondary calendar.
Calendar_Patch	Updates metadata for a calendar. This method supports patch semantics.
Calendar_Update	Updates metadata for a calendar.

Method	Description
Channel_Stop	Stop watching resources through this channel.

Method	Description
Color_Get	Returns the color definitions for calendars and events.

Method	Description
Event_Delete	Deletes an event.
Event_Get	Returns an event.
Event_Import	Imports an event. This operation is used to add a private copy of an existing event to a calendar.
Event_Insert	Creates an event.
Event_Instances	Returns instances of the specified recurring event.
Event_List	Returns events on the specified calendar.
Event_Move	Moves an event to another calendar, i.e. changes an event's organizer.
Event_Patch	Updates an event. This method supports patch semantics. The field values you specify replace the existing values. Fields that you don't specify in the request remain unchanged. Array fields, if

	specified, overwrite the existing arrays; this discards any previous array elements.
Event_QuickAdd	Creates an event based on a simple text string.
Event_Update	Updates an event.
Event_Watch	Watch for changes to Events resources.

Method	Description
Freebusy_Query	Returns free/busy information for a set of calendars.

Method	Description
Settings_Get	Returns a single user setting.
Settings_List	Returns all user settings for the authenticated user.
Settings_Watch	Watch for changes to Settings resources.

Switch Account

If you want to switch from one account to another, first call the method **Clear**, to erase the current calendar session and configure the new session.

Google Calendar | Load Calendars

The process to get all calendars from your account is very easy. Just follow the next steps:

1. Call the method **LoadCalendars**.
2. If the method returns True, then you can access the **Calendars** property and iterate over the list to get access to all Calendars.

```
TsgcHTTPGoogleCloud_Calendar_Client oGoogleCalendar = new TsgcHTTPGoogleCloud_Calendar_Client(NULL);
// ... configure OAuth2 options
oGoogleCalendar->GoogleCloudOptions->OAuth2->ClientId = "google ClientId";
oGoogleCalendar->GoogleCloudOptions->OAuth2->ClientSecret = "google ClientSecret";
// ... request calendars
if (oGoogleCalendar->LoadCalendars)
{
  // ... get calendars data
  for (int i = 0; i < oGoogleCalendar->Calendars->Count; i++)
  {
    vCalendarTitle = oGoogleCalendar->Calendars->Calendar[i]->Summary;
  }
}
else
{
  throw Exception("Error Calendar Sync");
}
```

Google Calendar | Sync Events

The process to get all events from a calendar is very easy. Just follow the next steps:

1. Call the method **LoadEvents** and pass the **CalendarId** as parameter**.**
2. If the method returns True, then you can access the **Calendars.Events** property and iterate over the list to get access to all Events of the calendar.

```
TsgcHTTPGoogleCloud_Calendar_Client oGoogleCalendar = new TsgcHTTPGoogleCloud_Calendar_Client(NULL);
// ... configure OAuth2 options
oGoogleCalendar->GoogleCloudOptions->OAuth2->ClientId = "google ClientId";
oGoogleCalendar->GoogleCloudOptions->OAuth2->ClientSecret = "google ClientSecret";
// ... request calendars first;
oGoogleCalendar->LoadCalendars;
// ... request events from first calendar
oCalendar = TsgcGoogleCalendarItem(oGoogleCalendar->Calendars->Calendar[0]);
if (oGoogleCalendar->LoadEvents(oCalendar->ID))
{
  // ... get events data
  for (int i = 0; i < oCalendar->Events->Count; i++)
  {
    vEventTitle = oCalendar->Events[i]->Summary;
  }
}
else
{
  throw Exception("Error Event Sync");
}
```

Google Calendar | RefreshToken

The Google Calendar API uses OAuth2 to authenticate against Google servers. sgcWebSockets has a component that handles the entire authentication process, but if your application closes and you attempt to connect again, you have two options:

1. Authenticate again using your Google APIs
2. Use the Refresh Token (if still valid), so you avoid the authentication process.

Using RefreshToken

The first time you authenticate, use the OnAuthToken event to save the **RefreshToken** if it exists. You can save it in an INI file, for example:

```
void OnGoogleCalendarAuthToken(TObject *Sender, string TokenType, string Token, string Data)
{
  TsgcJSON *oJSON = new TsgcJSON();
  try
  {
    oJSON->Read(Data);
    if (oJSON->Node["refresh_token"] != NULL)
    {
      TINIFile *oINI = new TINIFile(ChangeFileExt(Application->ExeName, ".ini"));
      try
      {
        oINI->WriteString("OAUTH2", "Token", oJSON->Node["refresh_token"]->Value);
      }
      __finally
      {
        oINI->Free();
      }
    }
  }
  __finally
  {
    oJSON->Free();
  }
}
```

Then, when you start your application again, if there is a RefreshToken, call the RefreshToken method and pass the token as an argument (you must first set the Google Calendar API keys). If successful, you will log in to Google servers without having to re-authenticate.

```
GoogleCalendar->RefreshToken("your refresh token here");
```

Google Calendar | Service Account

The Google Calendar client can work as a service without user interaction, so this is useful when you want to run a Windows service, a daemon, etc.

Google Cloud requires creating a **Service Account** (instead of OAuth2 credentials) to run this type of project, and the Google Calendar API requires that the service account uses **Domain-Wide Delegation** to get the required credentials to access the calendars.

You can read more about how to create Google Service Accounts.

Once the Google Cloud account has been configured with a service account and linked to a workspace email account using Domain-Wide Delegation, you can configure the Google Calendar client to work with it by following the next steps:

- Set the property **GoogleCloudOptions.Authentication** the value **gcaJWT**.
- Import the JSON file generated in your Google Cloud Account using the method **LoadSettingsFromFile**. This file contains the private key to encrypt the JWT and some other properties required by the client.
- After importing the JSON file, the following properties are automatically filled:
  - **ClientEmail**: is the service account name
  - **PrivateKeyId**: is the id of the private key file
  - **PrivateKey**: is the private key file
- Finally, set in the property **GoogleCloudOptions.JWT.Subject** the Workspace email account linked to the service account.

After configuring the client, you can start sending requests to the Google Calendar API without user interaction.

Google Cloud FCM

Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably send messages at no cost.

Using FCM, you can notify a client app that new email or other data is available to sync. You can send notification messages to drive user re-engagement and retention. For use cases such as instant messaging, a message can transfer a payload of up to 4096 bytes to a client app.

The component supports the HTTP v1 API.

Authorization

Google FCM component client can login to Google Servers using the following methods:

- **gcaOAuth2:** OAuth2 protocol
- **gcaJWT:** JWT tokens.

OAuth2

The login is done using a web browser where the user logs in with their own account and authorizes the FCM requests.

- **GoogleCloudOptions.OAuth2.ClientId:** is the ClientID provided by Google to Authenticate through OAuth2 protocol.
- **GoogleCloudOptions.OAuth2.ClientSecret:** is the Client Secret string provided by Google to Authenticate through OAuth2 protocol.
- **GoogleCloudOptions.OAuth2.Scope:** is the scope of OAuth2, usually there is no need to modify the default value unless you need to get more access than default.
- **GoogleCloudOptions.OAuth2.LocalIP:** the OAuth2 protocol requires a local server listening for the response from the authentication server. This is the IP or DNS name. By default, it is 127.0.0.1.
- **GoogleCloudOptions.OAuth2.LocalPort:** Local server listening port.
- **GoogleCloudOptions.OAuth2.RedirectURL:** if you need to set a redirect URL different from LocalPort + LocalIP, you can set it in this property (example: http://127.0.0.1:8080/oauth2).

Service Accounts

The login is done by signing the requests using a private key provided by Google. This method is recommended for automated services or applications without user interaction.

- **GoogleCloudOptions.JWT.ClientEmail:** the client email name provided when creating the new service account. "client_email" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.Subject:** the client email name provided when creating the new service account. "client_email" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKeyId:** the Private Key ID provided by Google. "private_key_id" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKey:** the Private Key certificate provided by Google. "private_key" node in the JSON configuration file.

TLS Options

Firebase Cloud Messaging connections can be tuned through the **TLSOptions** property.

- **TLSOptions.IOHandler:** choose the TLS stack that will be used (OpenSSL, SChannel...).
- **TLSOptions.Version:** request a specific TLS protocol version required by your project.
- **TLSOptions.VerifyCertificate:** enforce server certificate validation when connecting to Google.
- **TLSOptions.OpenSSL_Options.LibPath:** define where the OpenSSL runtime libraries are deployed.
- **TLSOptions.SChannel_Options.UseLegacyCredentials:** enable it when Windows SChannel requires legacy credential negotiation.

Example configuration:

```
TsgcHTTPGoogleCloud_FCM_Client *oFCM = new TsgcHTTPGoogleCloud_FCM_Client(NULL);
oFCM->TLSOptions->IOHandler = iohOpenSSL;
oFCM->TLSOptions->Version = tls1_3;
oFCM->TLSOptions->VerifyCertificate = true;
oFCM->TLSOptions->OpenSSL_Options->LibPath = oslpDefaultFolder;
```

When a new service account is created, you can download a JSON file with all configurations. This file can be processed by the FCM component, just call the method **LoadSettingsFromFile** and pass the JSON filename as argument.

Most common uses

- **Configuration**
  - **Google OAuth2 Keys**
  - **Service Accounts**

Google FCM Client

OAuth2

In order to work with the Google FCM API, the sgcWebSockets FCM component uses OAuth2 as the default authentication method, so first you must set your **ClientId** and **ClientSecret** from your Google account.

```
oFCM = new TsgcHTTPGoogleCloud_FCM_Client();
oFCM->GoogleCloudOptions->Authorization = gcaOAuth2;
oFCM->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oFCM->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
```

Service Accounts

Service Accounts require building a JWT and passing it as an authorization token.

```
TsgcHTTPGoogleCloud_FCM_Client *oFCM = new TsgcHTTPGoogleCloud_FCM_Client(this);
oFCM->GoogleCloudOptions->Authorization = gcaJWT;
oFCM->GoogleCloudOptions->JWT->ClientEmail = "...google email...";
oFCM->GoogleCloudOptions->JWT->Subject = "...google email...";
oFCM->GoogleCloudOptions->JWT->PrivateKeyId = "...private key id...";
oFCM->GoogleCloudOptions->JWT->PrivateKey->Lines->Text = "...private key certificate...";
```

This is required in order to get an Authorization Token Key from Google which will be used for all Rest API calls.

All methods return a response, which may be successful or return an error.

Send Message

Use the method SendMessage to send a notification message. The function has 2 arguments:

1. Project: is the project id of your google cloud account.
2. Payload: is the json text message. Example:

```
{
  "message": {
    "topic": "news",
    "notification": {
      "title": "Breaking News",
```

```
      "body": "New news story available."
    },
    "data": {
      "story_id": "story_12345"
    }
  }
}
```

TsgcWebView2

TsgcWebView2 is a visual VCL component that wraps Microsoft Edge WebView2 (Chromium). Drop it on a form to embed a modern, full-featured web browser in your Delphi application. It supports Delphi 7 through Delphi 13.

Requirements

- Microsoft Edge WebView2 Runtime (included with Windows 10/11 or downloadable from Microsoft)
- WebView2Loader.dll (included, place next to your executable)
- Windows only

Quick Start

Drop TsgcWebView2 on a form and navigate to a URL:

```
sgcWebView21 := TsgcWebView2.Create(Self);
sgcWebView21.Parent := Self;
sgcWebView21.Align := alClient;
sgcWebView21.DefaultURL := 'https://www.example.com';
```

Features

- **Navigation**
  - **Navigate, NavigateToString, GoBack, GoForward, Reload, Stop, POST navigation**
- **JavaScript**
  - **ExecuteScript (async), ExecuteScriptSync, AddInitScript**
- **Cookie Management**
  - **GetCookies, AddOrUpdateCookie, DeleteCookie, DeleteAllCookies**
- **Download Control**
  - **OnDownloadStarting, OnDownloadProgress, OnDownloadCompleted**
- **Settings & Configuration**
  - **ScriptEnabled, DevToolsEnabled, ContextMenuEnabled, and more**
- **Advanced Features**
  - **Print, Audio/Mute, Certificates, Favicon, Virtual Host, Screenshot**

Properties

- **DefaultURL**: string — URL to navigate to after initialization.
- **AutoInitialize**: Boolean — when True, the WebView2 is created automatically when the window handle is allocated. Default: True.
- **ZoomFactor**: Double — zoom level (1.0 = 100%).
- **UserDataFolder**: string — path for cookies, cache, and permissions storage.
- **BrowserExecutableFolder**: string — path to a fixed-version WebView2 Runtime distribution.
- **Settings**: TsgcWebView2Settings — design-time browser settings (ScriptEnabled, DevToolsEnabled, etc.).
- **URL**: string (read-only) — current page URL.
- **DocumentTitle**: string (read-only) — current page title.
- **CanGoBack**: Boolean (read-only) — whether back navigation is possible.
- **CanGoForward**: Boolean (read-only) — whether forward navigation is possible.
- **Initialized**: Boolean (read-only) — whether WebView2 is ready.
- **IsMuted**: Boolean — mute/unmute audio.
- **IsDocumentPlayingAudio**: Boolean (read-only) — whether the page is playing audio.
- **FaviconURI**: string (read-only) — URI of the current page favicon.
- **StatusBarText**: string (read-only) — current status bar text.
- **CookieManager**: TsgcWebView2CookieManager (read-only) — cookie management object.
- **WebView**: ICoreWebView2 (read-only) — direct COM interface access.
- **Controller**: ICoreWebView2Controller (read-only) — direct COM interface access.
- **Environment**: ICoreWebView2Environment (read-only) — direct COM interface access.

Methods

- **Navigate**(aURL: string) — navigate to a URL.
- **NavigateToString**(aHTML: string) — load HTML content directly.
- **GoBack** / **GoForward** — navigate history.
- **Reload** / **Stop** — reload or stop loading.
- **ExecuteScript**(aScript: string) — execute JavaScript asynchronously (result in OnScriptExecuted).
- **ExecuteScriptSync**(aScript: string): string — execute JavaScript synchronously, returns JSON result.
- **AddInitScript**(aScript: string) — add JavaScript that runs on every page load.
- **RemoveInitScript**(aScriptId: string) — remove a previously added init script.
- **NavigateWithPostData**(aURI, aMethod, aPostData, aHeaders: string) — navigate with custom HTTP method, body and headers.
- **PrintToPdf**(aFilePath: string) — save the page as PDF.
- **ShowPrintUI** — show the native print dialog.
- **CapturePreviewToFile**(aFilePath: string; aFormat: Integer) — screenshot (0=PNG, 1=JPEG).
- **OpenDevToolsWindow** — open Edge DevTools.
- **OpenTaskManagerWindow** — open Edge task manager.
- **SetVirtualHostNameToFolderMapping**(aHostName, aFolderPath: string; aAccessKind: Integer) — map hostname to local folder.
- **ClearVirtualHostNameToFolderMapping**(aHostName: string) — remove mapping.
- **PostWebMessageAsString**(aMessage: string) — send message to web content.
- **PostWebMessageAsJson**(aJson: string) — send JSON to web content.
- **GetProfileName**: string — get the browser profile name.
- **ClearBrowsingData**(aKinds: Cardinal) — clear specific browsing data types.
- **ClearAllBrowsingData** — clear all browsing data.
- **PostSharedBufferToScript**(aSharedBuffer: IUnknown; aAccess: Integer; aAdditionalDataAsJson: string) — share memory buffer with web content.
- **InitializeWebView** / **FinalizeWebView** — manual initialization control.

Events

- **OnInitialized** — fired when WebView2 is ready.
- **OnNavigationStarting**(aURI; aIsUserInitiated, aIsRedirected: Boolean; var aCancel: Boolean) — before navigation.
- **OnNavigationCompleted**(aIsSuccess: Boolean; aWebErrorStatus: Integer) — after navigation.
- **OnSourceChanged**(aIsNewDocument: Boolean) — URL changed.
- **OnDocumentTitleChanged**(aTitle: string) — page title changed.
- **OnWebMessageReceived**(aSource, aWebMessageAsJson, aWebMessageAsString: string) — message from web content.
- **OnNewWindowRequested**(aURI: string; aIsUserInitiated: Boolean; var aHandled: Boolean) — popup/new tab requested.
- **OnScriptExecuted**(aErrorCode: HRESULT; aResultAsJson: string) — async JS result.
- **OnContentLoading**(aIsErrorPage: Boolean) — content started loading.
- **OnHistoryChanged** — navigation history changed.
- **OnProcessFailed**(aKind: Integer) — browser process crashed.
- **OnPermissionRequested**(aURI: string; aKind: Integer; aIsUserInitiated: Boolean; var aState: Integer) — permission request.
- **OnWindowCloseRequested** — window.close() called.
- **OnDownloadStarting**(aURI, aResultFilePath: string; var aCancel, aHandled: Boolean; var aFilePath: string) — download started.
- **OnDownloadProgress**(aBytesReceived, aTotalBytes: Int64) — download progress.
- **OnDownloadCompleted**(aFilePath: string; aState: Integer) — download finished.
- **OnContextMenuRequested**(aMenuItems: string; aContextKind: Integer; aLocation: TPoint; var aHandled: Boolean) — right-click menu.
- **OnFaviconChanged**(aFaviconURI: string) — page icon changed.
- **OnClientCertificateRequested**(aHost: string; aPort: Integer; var aHandled: Boolean) — client cert needed.
- **OnServerCertificateError**(aRequestURI: string; aErrorStatus: Integer; var aAction: Integer) — cert error.
- **OnStatusBarTextChanged**(aText: string) — status bar text changed.
- **OnDOMContentLoaded** — DOM ready.
- **OnBasicAuthRequested**(aURI: string; var aUserName, aPassword: string; var aHandled: Boolean) — HTTP basic auth.
- **OnCapturePreviewCompleted**(aErrorCode: HRESULT) — screenshot finished.

TsgcWebView2 | Navigation

TsgcWebView2 provides several methods for navigating to URLs, loading HTML content, and controlling browser history.

Basic Navigation

Use the **Navigate** method to load a URL. The component fires OnNavigationStarting before the request and OnNavigationCompleted when it finishes.

```
// Navigate to a URL
sgcWebView21.Navigate('https://www.example.com');
```

You can also set the **DefaultURL** property at design time or before initialization. The component navigates to this URL automatically after WebView2 is ready.

```
// Set the default URL before initialization
sgcWebView21.DefaultURL := 'https://www.example.com';
```

HTML Content

Use **NavigateToString** to load HTML content directly without a server or file.

```
// Load HTML content directly
sgcWebView21.NavigateToString(
  '<html><body><h1>Hello from Delphi</h1>' +
  '<p>This content was loaded with NavigateToString.</p>' +
  '</body></html>');
```

History Navigation

Use **GoBack** and **GoForward** to navigate the browser history. Check **CanGoBack** and **CanGoForward** before calling these methods.

```
// Navigate back
if sgcWebView21.CanGoBack then
  sgcWebView21.GoBack;

// Navigate forward
if sgcWebView21.CanGoForward then
  sgcWebView21.GoForward;
```

Use **Reload** to refresh the current page and **Stop** to cancel a pending navigation.

```
// Reload the current page
sgcWebView21.Reload;

// Stop loading
sgcWebView21.Stop;
```

POST Navigation

Use **NavigateWithPostData** to send an HTTP request with a custom method, body, and headers. This is useful for submitting form data or calling REST APIs directly in the browser.

```
// POST form data with custom headers
sgcWebView21.NavigateWithPostData(
  'https://api.example.com/login',
```

```
  'POST',
  'username=admin&password=secret',
  'Content-Type: application/x-www-form-urlencoded');
```

```
// POST JSON data
sgcWebView21.NavigateWithPostData(
  'https://api.example.com/data',
  'POST',
  '{"name":"John","age":30}',
  'Content-Type: application/json');
```

Navigation Events

Use **OnNavigationStarting** to inspect or cancel a navigation before it begins. Set **aCancel** to True to block the request.

```
procedure TFormMain.sgcWebView21NavigationStarting(Sender: TObject;
  const aURI: string; aIsUserInitiated, aIsRedirected: Boolean;
  var aCancel: Boolean);
begin
  // Block navigation to unwanted domains
  if Pos('ads.example.com', aURI) > 0 then
    aCancel := True;
end;
```

Use **OnNavigationCompleted** to check whether the navigation succeeded or failed.

```
procedure TFormMain.sgcWebView21NavigationCompleted(Sender: TObject;
  aIsSuccess: Boolean; aWebErrorStatus: Integer);
begin
  if aIsSuccess then
    StatusBar1.SimpleText := 'Page loaded: ' + sgcWebView21.URL
  else
    StatusBar1.SimpleText := 'Navigation failed, error: ' +
      IntToStr(aWebErrorStatus);
end;
```

New Window Handling

When the page opens a popup or a link with target="_blank", the **OnNewWindowRequested** event fires. Set **aHandled** to True to prevent the default popup and navigate in the same window instead.

```
procedure TFormMain.sgcWebView21NewWindowRequested(Sender: TObject;
  const aURI: string; aIsUserInitiated: Boolean;
  var aHandled: Boolean);
begin
  // Navigate in the same window instead of opening a popup
  aHandled := True;
  sgcWebView21.Navigate(aURI);
end;
```

TsgcWebView2 | JavaScript

TsgcWebView2 provides methods to execute JavaScript in the browser context, retrieve results, and establish two-way communication between your Delphi application and web content.

Async Execution

Use **ExecuteScript** to run JavaScript asynchronously. The result is returned in the **OnScriptExecuted** event as a JSON string.

```
// Execute JavaScript asynchronously
sgcWebView21.ExecuteScript('document.title');
```

```
procedure TFormMain.sgcWebView21ScriptExecuted(Sender: TObject;
  aErrorCode: HRESULT; const aResultAsJson: string);
begin
  if aErrorCode = S_OK then
    ShowMessage('Result: ' + aResultAsJson)
  else
    ShowMessage('Script error: ' + IntToStr(aErrorCode));
end;
```

You can execute any valid JavaScript expression. The return value is always serialized as JSON.

```
// Get the number of links on the page
sgcWebView21.ExecuteScript('document.querySelectorAll("a").length');

// Modify page content
sgcWebView21.ExecuteScript(
  'document.body.style.backgroundColor = "lightyellow"');
```

Sync Execution

Use **ExecuteScriptSync** for blocking execution that returns the result directly. This is simpler when you need the value immediately, but it blocks the calling thread until the script completes.

```
var
  vResult: string;
begin
  // Get the page title synchronously
  vResult := sgcWebView21.ExecuteScriptSync('document.title');
  ShowMessage('Title: ' + vResult);

  // Get form field value
  vResult := sgcWebView21.ExecuteScriptSync(
    'document.getElementById("email").value');
  ShowMessage('Email: ' + vResult);
end;
```

Init Scripts

Use **AddInitScript** to register JavaScript that runs automatically on every page load, before any other scripts on the page. This is useful for injecting polyfills, overriding browser APIs, or setting up message handlers.

```
// Add a script that runs on every page load
sgcWebView21.AddInitScript(
  'window.addEventListener("DOMContentLoaded", function() {' +
  '  console.log("Page loaded at: " + new Date().toISOString());' +
  '});');
```

```
// Override window.alert to send messages to Delphi
sgcWebView21.AddInitScript(
  'window.alert = function(msg) {' +
  '  window.chrome.webview.postMessage(msg);' +
  '};');
```

Use **RemoveInitScript** to unregister a previously added init script by its ID.

Web Messaging

Web messaging provides two-way communication between your Delphi application and JavaScript running in the browser. Use **PostWebMessageAsString** or **PostWebMessageAsJson** to send data from Delphi to JavaScript, and handle **OnWebMessageReceived** to receive data from JavaScript.
Sending messages from Delphi to JavaScript:

```
// Send a plain string message
sgcWebView21.PostWebMessageAsString('Hello from Delphi!');

// Send a JSON message
sgcWebView21.PostWebMessageAsJson('{"action":"refresh","id":42}');
```

Receiving messages in JavaScript:

```
// Add an init script to listen for messages from Delphi
sgcWebView21.AddInitScript(
  'window.chrome.webview.addEventListener("message", function(e) {' +
  '  console.log("Received from Delphi: " + e.data);' +
  '});');
```

Receiving messages from JavaScript in Delphi:

```
procedure TFormMain.sgcWebView21WebMessageReceived(Sender: TObject;
  const aSource, aWebMessageAsJson, aWebMessageAsString: string);
begin
  // aWebMessageAsString contains the plain text message
  // aWebMessageAsJson contains the JSON-serialized message
  Memo1.Lines.Add('Message from web: ' + aWebMessageAsString);
end;
```

From JavaScript, send messages to Delphi using:

```
// JavaScript code inside the web page:
// window.chrome.webview.postMessage('Hello from JavaScript!');
// window.chrome.webview.postMessage({action: 'save', data: [1,2,3]});
```

TsgcWebView2 | Cookie Management

TsgcWebView2 exposes a **CookieManager** property for reading, creating, updating, and deleting cookies in the browser context.

Getting Cookies

Use **CookieManager.GetCookies** to retrieve cookies for a given URI. The method returns a list of cookie objects that you can iterate.

```
procedure TFormMain.ButtonGetCookiesClick(Sender: TObject);
var
  vCookies: TsgcWebView2CookieList;
  i: Integer;
begin
  vCookies := sgcWebView21.CookieManager.GetCookies(
    'https://www.example.com');
  try
    for i := 0 to vCookies.Count - 1 do
      Memo1.Lines.Add(
        vCookies[i].Name + ' = ' + vCookies[i].Value);
  finally
    vCookies.Free;
  end;
end;
```

Adding Cookies

Use **CookieManager.AddOrUpdateCookie** to create a new cookie or update an existing one. Specify the name, value, domain, and path at minimum.

```
// Add a session cookie
sgcWebView21.CookieManager.AddOrUpdateCookie(
  'session_id',             // name
  'abc123def456',           // value
  '.example.com',           // domain
  '/');                     // path
```

```
// Add a cookie with an expiration date
sgcWebView21.CookieManager.AddOrUpdateCookie(
  'preferences',            // name
  'theme=dark',             // value
  '.example.com',           // domain
  '/',                      // path
  Now + 30);                // expires in 30 days
```

Deleting Cookies

Use **DeleteCookie** to remove a specific cookie by name and URI. Use **DeleteAllCookies** to clear all cookies from the browser.

```
// Delete a specific cookie
sgcWebView21.CookieManager.DeleteCookie(
  'session_id',
  'https://www.example.com');
```

```
// Delete cookies matching a domain and path
sgcWebView21.CookieManager.DeleteCookiesWithDomainAndPath(
  'session_id',
  '.example.com',
  '/');
```

```
// Delete all cookies
sgcWebView21.CookieManager.DeleteAllCookies;
```

TsgcWebView2 | Download Control

TsgcWebView2 provides events to intercept, monitor, and control file downloads initiated by the browser.

Download Events

The **OnDownloadStarting** event fires when a download begins. You can cancel the download, change the destination file path, or let it proceed with the default behavior.

```
procedure TFormMain.sgcWebView21DownloadStarting(Sender: TObject;
  const aURI, aResultFilePath: string;
  var aCancel, aHandled: Boolean; var aFilePath: string);
begin
  // Log the download
  Memo1.Lines.Add('Download starting: ' + aURI);
  Memo1.Lines.Add('Default path: ' + aResultFilePath);

  // Cancel downloads of .exe files
  if Pos('.exe', LowerCase(aURI)) > 0 then
  begin
    aCancel := True;
    ShowMessage('Executable downloads are blocked.');
  end;
end;
```

Download Progress

The **OnDownloadProgress** event fires periodically during the download, reporting bytes received and total bytes expected.

```
procedure TFormMain.sgcWebView21DownloadProgress(Sender: TObject;
  aBytesReceived, aTotalBytes: Int64);
begin
  if aTotalBytes > 0 then
    ProgressBar1.Position :=
      Round((aBytesReceived / aTotalBytes) * 100)
  else
    ProgressBar1.Position := 0;

  LabelStatus.Caption := Format('Downloaded %d of %d bytes',
    [aBytesReceived, aTotalBytes]);
end;
```

Download Complete

The **OnDownloadCompleted** event fires when the download finishes. Check **aState** to determine whether the download succeeded, was cancelled, or failed.

```
procedure TFormMain.sgcWebView21DownloadCompleted(Sender: TObject;
  const aFilePath: string; aState: Integer);
begin
  case aState of
    0: // Completed
      ShowMessage('Download complete: ' + aFilePath);
    1: // Cancelled
      ShowMessage('Download was cancelled.');
    2: // Failed
      ShowMessage('Download failed.');
  end;
end;
```

Custom Download Path

Set the **aFilePath** parameter in OnDownloadStarting to redirect the download to a custom location. Set **aHandled** to True to suppress the default save dialog.

```
procedure TFormMain.sgcWebView21DownloadStarting(Sender: TObject;
  const aURI, aResultFilePath: string;
  var aCancel, aHandled: Boolean; var aFilePath: string);
begin
  // Redirect all downloads to a custom folder
  aFilePath := 'C:\Downloads\' + ExtractFileName(aResultFilePath);
  aHandled := True; // suppress the default save dialog
end;
```

TsgcWebView2 | Settings

TsgcWebView2 exposes a **Settings** property of type TsgcWebView2Settings that controls browser behavior. These settings can be configured at design time or changed at runtime after initialization.

TsgcWebView2Settings Properties

- **ScriptEnabled**: Boolean — enable or disable JavaScript execution. Default: True.
- **WebMessageEnabled**: Boolean — enable or disable web messaging (postMessage). Default: True.
- **DefaultScriptDialogsEnabled**: Boolean — enable or disable default JavaScript dialogs (alert, confirm, prompt). Default: True.
- **StatusBarEnabled**: Boolean — show or hide the browser status bar. Default: True.
- **DevToolsEnabled**: Boolean — allow or block access to Edge DevTools. Default: True.
- **ContextMenuEnabled**: Boolean — enable or disable the browser right-click context menu. Default: True.
- **ZoomControlEnabled**: Boolean — allow or block user-initiated zoom (Ctrl+scroll, Ctrl+plus/minus). Default: True.
- **BuiltInErrorPageEnabled**: Boolean — show or hide the built-in error page for navigation failures. Default: True.

Design-Time Configuration

Set the Settings properties in the Object Inspector. They are applied automatically after the WebView2 environment is initialized.

```
// These values can also be set in the Object Inspector
sgcWebView21.Settings.ScriptEnabled := True;
sgcWebView21.Settings.DevToolsEnabled := False;
sgcWebView21.Settings.ContextMenuEnabled := False;
sgcWebView21.Settings.ZoomControlEnabled := False;
```

Runtime Configuration

You can change settings at runtime after the component is initialized. Changes take effect immediately.

```
procedure TFormMain.sgcWebView21Initialized(Sender: TObject);
begin
  // Disable right-click menu and DevTools at runtime
  sgcWebView21.Settings.ContextMenuEnabled := False;
  sgcWebView21.Settings.DevToolsEnabled := False;

  // Disable JavaScript dialogs
  sgcWebView21.Settings.DefaultScriptDialogsEnabled := False;
end;
```

UserDataFolder

The **UserDataFolder** property specifies the directory where the browser stores cookies, cache, permissions, and other profile data. Each unique folder creates an isolated browser profile.

```
// Use a custom data folder for isolated profiles
sgcWebView21.UserDataFolder := 'C:\AppData\MyApp\Profile1';
```

Set this property before initialization (before the component creates the WebView2 environment). If not set, a default folder is used in the application's temporary directory.

BrowserExecutableFolder

The **BrowserExecutableFolder** property points to a fixed-version WebView2 Runtime distribution. Use this to ship a specific browser version with your application instead of relying on the system-installed runtime.

```
// Use a fixed-version runtime bundled with the application
sgcWebView21.BrowserExecutableFolder :=
  ExtractFilePath(ParamStr(0)) + 'WebView2Runtime';
```

Set this property before initialization. If left empty, the component uses the system-installed Evergreen WebView2 Runtime.

TsgcWebView2 | Advanced Features

TsgcWebView2 provides access to advanced browser capabilities including printing, screenshots, audio control, certificate handling, and more.

Print Support

Use **PrintToPdf** to save the current page as a PDF file, or **ShowPrintUI** to display the native print dialog.

```
// Save the current page as PDF
sgcWebView21.PrintToPdf('C:\Output\page.pdf');
```

```
// Show the native print dialog
sgcWebView21.ShowPrintUI;
```

Screenshot Capture

Use **CapturePreviewToFile** to take a screenshot of the current page. The format parameter specifies the image type: 0 for PNG, 1 for JPEG. The **OnCapturePreviewCompleted** event fires when the capture finishes.

```
// Capture as PNG
sgcWebView21.CapturePreviewToFile('C:\Output\screenshot.png', 0);

// Capture as JPEG
sgcWebView21.CapturePreviewToFile('C:\Output\screenshot.jpg', 1);
```

```
procedure TFormMain.sgcWebView21CapturePreviewCompleted(
  Sender: TObject; aErrorCode: HRESULT);
begin
  if aErrorCode = S_OK then
    ShowMessage('Screenshot saved.')
  else
    ShowMessage('Screenshot failed: ' + IntToStr(aErrorCode));
end;
```

Audio / Mute Control

Use the **IsMuted** property to mute or unmute audio playback. Check **IsDocumentPlayingAudio** to detect whether the page is currently playing audio.

```
// Toggle mute
sgcWebView21.IsMuted := not sgcWebView21.IsMuted;

// Check if audio is playing
if sgcWebView21.IsDocumentPlayingAudio then
  LabelStatus.Caption := 'Audio is playing';
```

Certificate Handling

Use **OnClientCertificateRequested** to respond when a server requires a client certificate. Use **OnServerCertificateError** to handle TLS certificate errors (for example, self-signed certificates in development).

```
procedure TFormMain.sgcWebView21ClientCertificateRequested(
  Sender: TObject; const aHost: string; aPort: Integer;
  var aHandled: Boolean);
begin
  // Handle client certificate selection
  aHandled := True;
```

```
end;
```

```
procedure TFormMain.sgcWebView21ServerCertificateError(
  Sender: TObject; const aRequestURI: string;
  aErrorStatus: Integer; var aAction: Integer);
begin
  // Accept self-signed certificates in development
  // aAction: 0 = deny, 1 = allow
  aAction := 1;
end;
```

Virtual Host Mapping

Use **SetVirtualHostNameToFolderMapping** to map a hostname to a local folder. This lets web content reference local files using a virtual URL instead of file:// paths.

```
// Map "app.local" to a local folder
sgcWebView21.SetVirtualHostNameToFolderMapping(
  'app.local',
  'C:\MyApp\WebContent',
  0); // 0 = deny remote access

// Now navigate to local content using the virtual host
sgcWebView21.Navigate('https://app.local/index.html');
```

```
// Remove the mapping
sgcWebView21.ClearVirtualHostNameToFolderMapping('app.local');
```

Profile Management

Use **GetProfileName** to retrieve the current browser profile name. Use **ClearBrowsingData** or **ClearAllBrowsing-Data** to remove cached data, cookies, and other browsing artifacts.

```
// Get the profile name
ShowMessage('Profile: ' + sgcWebView21.GetProfileName);

// Clear all browsing data
sgcWebView21.ClearAllBrowsingData;

// Clear specific browsing data types
sgcWebView21.ClearBrowsingData($0001); // cache only
```

Basic Authentication

The **OnBasicAuthRequested** event fires when a server requests HTTP Basic authentication. Provide the credentials and set **aHandled** to True.

```
procedure TFormMain.sgcWebView21BasicAuthRequested(Sender: TObject;
  const aURI: string; var aUserName, aPassword: string;
  var aHandled: Boolean);
begin
  aUserName := 'admin';
  aPassword := 'secret';
  aHandled := True;
end;
```

Context Menu

The **OnContextMenuRequested** event fires when the user right-clicks in the browser. Set **aHandled** to True to suppress the default menu and show your own.

```
procedure TFormMain.sgcWebView21ContextMenuRequested(Sender: TObject;
  const aMenuItems: string; aContextKind: Integer;
  aLocation: TPoint; var aHandled: Boolean);
```

```
begin
  // Suppress the default context menu
  aHandled := True;

  // Show a custom popup menu at the click location
  PopupMenu1.Popup(aLocation.X, aLocation.Y);
end;
```

Favicon

The **FaviconURI** property returns the URI of the current page's favicon. The **OnFaviconChanged** event fires when the favicon changes.

```
procedure TFormMain.sgcWebView21FaviconChanged(Sender: TObject;
  const aFaviconURI: string);
begin
  LabelFavicon.Caption := 'Favicon: ' + aFaviconURI;
end;
```

Status Bar

The **StatusBarText** property returns the current status bar text (typically the URL of a hovered link). The **OnStatusBarTextChanged** event fires when the text changes.

```
procedure TFormMain.sgcWebView21StatusBarTextChanged(Sender: TObject;
  const aText: string);
begin
  StatusBar1.SimpleText := aText;
end;
```

Task Manager

Use **OpenTaskManagerWindow** to open the Edge browser task manager, which shows memory and CPU usage for each browser process.

```
sgcWebView21.OpenTaskManagerWindow;
```

Shared Buffer

Use **PostSharedBufferToScript** to share a memory buffer between your Delphi application and web content for high-performance data transfer.

```
// Share a buffer with web content
sgcWebView21.PostSharedBufferToScript(
  vSharedBuffer,  // IUnknown shared buffer object
  0,              // 0 = read-only, 1 = read-write
  '{"type":"image","width":640,"height":480}');
```

Direct COM Access

For advanced scenarios not covered by the component API, use the **WebView**, **Controller**, and **Environment** properties to access the underlying WebView2 COM interfaces directly.

```
var
  vWebView: ICoreWebView2;
begin
  vWebView := sgcWebView21.WebView;
  if Assigned(vWebView) then
  begin
    // Call any ICoreWebView2 method directly
  end;
end;
```

TsgcUDPCLient

TsgcUDPClient implements the UDP Client based on Indy library.

Introduction

TsgcUDPClient implements the UDP Client based on Indy library.

UDP is a connectionless protocol where there is no assurance that message sent arrive to the destination but opposite to TCP protocol, it's much faster.

1. Drop a **TsgcUDPClient** component onto the form

2. Set **Host** and **Port** (default is 80) to connect to an available UDP Server.

```
oClient = new TsgcUDPCLient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
```

3. You can connect through an HTTP Proxy Server, you need to define proxy properties:

Host: proxy server hostname.
Port: proxy server port number.
Username: username for authentication, leave blank for anonymous.
Password: password for authentication, leave blank for anonymous.

4. If you want, you can handle the events

5. Call **WriteData** method to send any message to the UDP server.

Reference

- Properties — 9 properties available on the component.
- Methods — 2 public methods.
- Events — 3 events you can handle.

TsgcUDPServer

TsgcUDPServer implements the UDP Server based on Indy library.

Introduction

TsgcUDPServer implements the UDP Server based on Indy library.

UDP it's a connection less protocol where there is no assurance that message sent arrive to the destination but opposite to TCP protocol, it's much faster.

1. Drop a **TsgcUDPServer** component onto the form

2. Set the listening **Port**.

```
oClient = new TsgcUDPServer();
oClient->Port = 80;
```

3. To **start** the server, set the property **Active = true**.

Reference

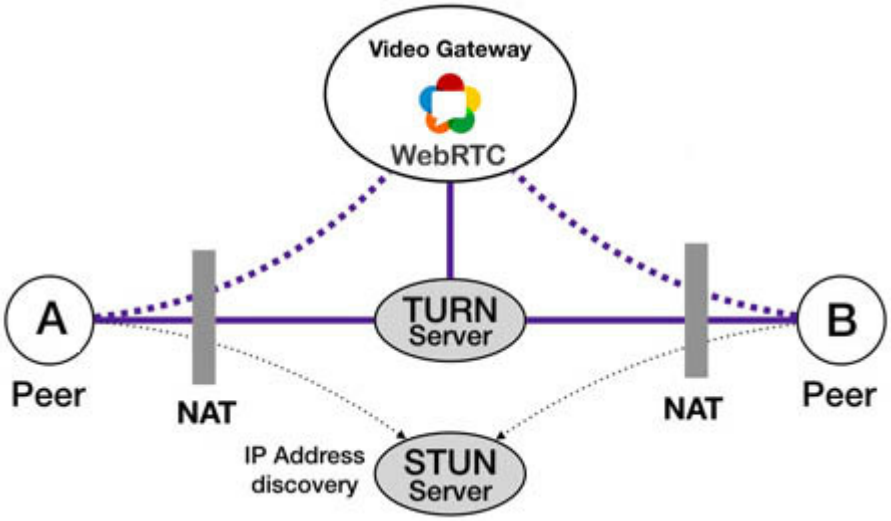
- Properties — 11 properties available on the component.
- Methods — 7 public methods.
- Events — 6 events you can handle.

STUN

STUN (Session Traversal Utilities for NAT) is an IETF protocol used for real-time audio and video in IP networks. STUN is a server-client protocol, a STUN server usually operates on both UDP and TCP and listens on port 3478.

The main purpose of the STUN protocol is to enable a device running behind a NAT to discover its public IP and what type of NAT it is behind.

STUN provides a mechanism to communicate between peers behind a NAT. The peers send a request to a STUN server to know which is the public IP address and Port. The binding requests sent from client to server are used to determine the IP and ports bindings allocated by NAT's. The STUN client sends a Binding request to the STUN server, the server examines the source IP and Port used by client, and returns this information to the client.



The STUN server basically sends two types of responses: successful or error. Every response has a list of attributes that contain information about the binding IP address, error code, reason for the error, etc.

Components

- **TsgcSTUNClient:** the client component that implements the STUN protocol and allows you to send binding requests to STUN servers.
- **TsgcSTUNServer:** the server component that implements the STUN protocol.

TsgcSTUNClient

TsgcSTUNClient component.

Introduction

TsgcSTUNClient is the client that implements the STUN protocol and allows you to send binding requests to STUN servers.

The component allows you to use **UDP** and **TCP** as transport. When using UDP as transport, it implements a **Re-transmission mechanism** to re-send requests if the response has not arrived after a short time.

Basic usage

Usually STUN servers run on UDP port 3478 and don't require authentication, so in order to send a STUN request binding, fill the server properties to allow the client to know where to connect and Handle the events where the component will receive the response from server.

Configure the server

- Host: the IP or DNS name of the server, example: stun.sgcwebsockets.com
- Port: the listening Server port, example: 3478

Call the method **SendRequest**, to send a request binding to STUN server.

Handle the events

- If the server returns a successful response, the event **OnSTUNResponseSuccess** will be called and you can access the binding information by reading the **aBinding** object.
- If the server returns an error, the event **OnSTUNResponseError** will be called and you can access the Error Code and Reason reading the **aError** object.

```
TsgcSTUNClient oSTUN = new TsgcSTUNClient(this);
oSTUN->Host = "stun.sgcwebsockets.com";
oSTUN->Port = 3478;
oSTUN->SendRequest();

private void OnSTUNResponseSuccess(TObject *Sender, const TsgcSocketConnection *aSocket,
  const TsgcSTUN_Message *aMessage, const TsgcSTUN_ResponseBinding *aBinding)
{
  DoLog("Remote IP: " + aBinding->RemoteIP + ". Remote Port: " + IntToStr(aBinding->RemotePort));
}

private void OnSTUNResponseError(TObject *Sender, const TsgcSocketConnection *aSocket,
  const TsgcSTUN_Message *aMessage, const TsgcSTUN_ResponseError *aError)
{
  DoLog("Error: " + IntToStr(aError->Code) + " " + aError->Reason);
}
```

Reference

- Properties — 9 properties available on the component.
- Methods — 3 public methods.
- Events — 4 events you can handle.

Guides

- **Bindings**
  - UDP Retransmissions
  - Long Term Credentials
  - Attributes

STUN Client | UDP Retransmissions

When running **STUN** over **UDP**, it's possible that the **STUN message** might be **dropped** by the network. Reliability of STUN request/response transactions is accomplished through retransmissions of the request message by the client application itself.

A client should retransmit a STUN request message starting with an interval of RTO ("Retransmission TimeOut"), doubling after each retransmission. The RTO is an estimate of the round-trip time.

By default, the sgcWebSockets STUN Client is already configured with a RTO of 500 ms and a Max Retries value of 7.

For example, assuming an RTO of 500 ms, requests would be sent at times 0 ms, 500 ms, 1500 ms, 3500 ms, 7500 ms, 15500 ms, and 31500 ms. If the client has not received a response after 39500 ms, the client will consider the transaction to have timed out.

```
TsgcSTUNClient oSTUN = new TsgcSTUNClient(this);
oSTUN->Host = "stun.sgcwebsockets.com";
oSTUN->Port = 3478;
oSTUN->RetransmissionOptions->Enabled = true;
oSTUN->RetransmissionOptions->RTO = 500;
oSTUN->RetransmissionOptions->MaxRetries = 7;
oSTUN->SendRequest();
```

STUN Client | Long Term Credentials

The long-term credential mechanism relies on a long-term credential, in the form of a username and password that are shared between client and server. The credential is considered long-term since it is assumed that it is provisioned for a user and remains in effect until the user is no longer a subscriber of the system or until it is changed.

You can configure the Long-term credentials in the sgcWebSockets STUN client using the following code.

```
TsgcSTUNClient oSTUN = new TsgcSTUNClient(this);
oSTUN->Host = "stun.sgcwebsockets.com";
oSTUN->Port = 3478;
oSTUN->STUNOptions->Authentication->Credentials = stauLongTermCredential;
oSTUN->STUNOptions->Authentication->Username = "user_name";
oSTUN->STUNOptions->Authentication->Password = "secret";
oSTUN->SendRequest();
```

If the server requires long-term credentials and the credentials sent by the client are wrong, the client will receive a 401 Unauthorized error as a response in the **OnSTUNResponseError** event.

STUN Client | Attributes

Every time a server sends a message to client, as a response message to a request binding, the STUN message contains a list of attributes with information about the response.

You can access these attributes using the TsgcSTUN_Message class and its Attributes property, which contains a list of TsgcSTUN_Attribute objects with useful information.

```
private void OnSTUNResponseSuccess(TObject * Sender, const TsgcSocketConnection *aSocket,
  const TsgcSTUN_Message *aMessage, const TsgcSTUN_ResponseBinding *aBinding)
{
  DoLog("#binding: " + aBinding->RemoteIP + ":" + IntToStr(aBinding->RemotePort));

  for (int i = 0; i < aMessage->Attributes->Count; i++)
  {
    switch (TsgcSTUN_Attribute(aMessage->Attributes->Items[i])->AttributeType)
      stmaFingerprint:
        DoLog("#fingerprint: " + IntToStr(dynamic_cast<TsgcSTUN_Attribute_FINGERPRINT*>
          (aMessage->Attributes->Items[i])->Fingerprint));
      stmaSoftware:
        DoLog("#software: " + dynamic_cast<TsgcSTUN_Attribute_SOFTWARE*>
          (aMessage->Attributes->Items[i])->Software);
      stmaResponse_Origin:
        DoLog("#response_origin: " + dynamic_cast<TsgcSTUN_Attribute_RESPONSE_ORIGIN*>
          (aMessage->Attributes->Items[i])->Address + ":" +
          IntToStr(dynamic_cast<TsgcSTUN_Attribute_RESPONSE_ORIGIN*>(aMessage->Attributes->Items
          [i])->Port));
    }
  }
}
```

TsgcSTUNServer

The STUN server can be configured with or without Authentication, can verify Fingerprint Attribute, send an alternate server and more.

Introduction

TsgcSTUNServer is the server that implements the STUN protocol and allows you to process binding requests from STUN clients.

The STUN server can be configured with or without Authentication, can verify Fingerprint Attribute, send an alternate server and more.

Basic usage

Usually STUN servers run on UDP port 3478 and don't require authentication, so in order to configure a STUN server, set the listening port (by default 3478) and start the server.

Configure the server

- Port: the listening Server port, example: 3478

Set the property **Active = True** to start the STUN server.

```
TsgcSTUNServer oSTUN = new TsgcSTUNServer();
oSTUN->Port = 3478;
oSTUN->Active = true;
```

Reference

- Properties — 8 properties available on the component.
- Methods — 2 public methods.
- Events — 4 events you can handle.

Guides

- **Configurations**
  - Long-Term Credentials
  - Alternate Server

STUN Server | Long-Term Credentials

Usually STUN Servers are configured without Authentication, so any STUN client can send a binding request and expect a response from server without Authentication.

sgcWebSockets STUN Server supports Long-Term Credentials, so you can configure TsgcSTUNServer to only allow binding requests with Long-Term credentials info.

To configure it, access to STUNOptions.Authorization property and enable it.
Then access to LongTermCredentials property and enabled it. By default, this type of authorization is already configured with a Realm string and with a default StaleNonce value of 10 minutes (= 600 seconds).

```
TsgcSTUNServer oSTUN = new TsgcSTUNServer();
oSTUN->STUNOptions->Authentication->Enabled = true;
oSTUN->STUNOptions->Authentication->LongTermCredentials->Enabled = true;
oSTUN->STUNOptions->Authentication->LongTermCredentials->Realm = "sgcWebSockets";
oSTUN->STUNOptions->Authentication->LongTermCredentials->StaleNonce = 600;
oSTUN->Port = 3478;
oSTUN->Active = true;

private void OnSTUNRequestAuthorization(TObject *Sender, const TsgcSTUN_Message *aRequest,
  const string aUsername, const string aRealm, ref string Password)
{
  if (aUsername == "my-user")
  {
    Password = "my-password";
  }
}
```

STUN Server | Alternate Server

The alternate server represents an alternate transport address identifying a different STUN server that the STUN client should try.

The STUN Server can be configured to send an alternate server as a response to a binding request, to configure this behaviour, just access to STUNOptions.BindingAttributes.AlternateServer property and configure here the values required.

```
TsgcSTUNServer oSTUN = new TsgcSTUNServer();
oSTUN->Port = 3478;
oSTUN->STUNOptions->BindingAttributes->AlternateServer->Enabled = true;
oSTUN->STUNOptions->BindingAttributes->AlternateServer->IPAddress = "80.54.54.1";
oSTUN->STUNOptions->BindingAttributes->AlternateServer->Port = 3478;
oSTUN->Active = true;
```

When the client receives the Alternate Server response attribute, it will try to send a request binding to the new server.

TURN

Traversal Using Relays around NAT (TURN) protocol enables a server to relay data packets between devices.

If the public IP address of both the caller and callee is not discovered, TURN provides a fallback technique to relay the call between endpoints.

Connecting a WebRTC session is an orchestrated effort done with the assistance of multiple WebRTC servers. The NAT traversal servers in WebRTC are in charge of making sure the media gets properly connected. These servers are STUN and TURN.

How WebRTC sessions connect

Directly

If both devices are on the local network, then no special effort is needed to get them connected to each other. If one device has the local IP address of the other device, then they can communicate with each other directly.

Directly with public IP Address

Connecting WebRTC directly using public IP address obtained via STUN protocol.

Route through a TURN Server

When peers are behind a NAT and there are Firewalls, direct connection is not possible, so a TURN server is required to route the data between the peers.

Components

- **TsgcTURNClient:** the client component that implements the TURN protocol and allows you to allocate, create permissions, send indications, etc. to a TURN server.
- **TsgcTURNServer:** the server component that implements the TURN protocol.

TsgcTURNClient

TsgcTURNClient component.

Introduction

TsgcTURNClient is the client that implements the TURN protocol and allows you to send allocation requests to TURN servers. The client inherits from STUN Client, so all methods supported by STUN client are already supported by TURN Client.

Basic usage

Usually TURN servers run on UDP port 3478 and don't require authentication, so in order to send a TURN request, fill the server properties to allow the client know where connect and Handle the events where the component will receive the response from server.

Configure the server

- Host: the IP or DNS name of the server, example: turn.sgcwebsockets.com
- Port: the listening Server port, example: 3478

Call the method **Allocate**, to send a request to allocate an IP Address and a Port to the TURN server.

Handle the events

- If the server returns a successful response, the event **OnTURNAllocateSuccess** will be called and you can access to the Allocation information reading the **aAllocation** object.
- If the server returns an error, the event **OnSTUNResponseError** will be called and you can access the Error Code and Reason reading the **aError** object.

```
TsgcTURNClient oTURN = new TsgcTURNClient(this);
oTURN->Host = "turn.sgcwebsockets.com";
oTURN->Port = 3478;
oTURN->Allocate();

private void OnTURNAllocate(TObject *Sender, const TsgcSocketConnection *aSocket,
  const TsgcSTUN_Message *aMessage, const TsgcTURN_ResponseAllocation *aAllocation)
{
  DoLog("Relayed IP: " + aAllocation->RelayedIP + ". Relayed Port: " + IntToStr(aAllocation->RelayedPort));
}

private void OnSTUNResponseError(TObject *Sender, const TsgcSTUN_Message *aMessage,
const TsgcSTUN_ResponseError *aError)
{
  DoLog("Error: " + IntToStr(aError->Code) + " " + aError->Reason);
}
```

Reference

- Properties — 10 properties available on the component.
- Methods — 9 public methods.
- Events — 11 events you can handle.

Guides

- **Allocation**

    - Allocate IP Address
    - Create Permissions
- **Indications**
    - Send Indication
- **Channels**
    - TURN Client Channels

TURN Client | Allocate IP Address

TURN Protocol allows you to use a Relayed IP Address to exchange data between peers that are behind NATs.

To create a new Relayed IP Address on a TURN server, the client must first call the method **Allocate**, this method sends a Request to the TURN server to create a new Relayed IP Address, if the TURN server can create a new Relayed IP Address, the client will receive a successful response. The client will be able to communicate with other peers during the time defined in the Allocation's lifetime.

```
TsgcTURNClient oTURN = new TsgcTURNClient(this);
oTURN->Host = "turn.sgcwebsockets.com";
oTURN->Port = 3478;
oTURN->Allocate();


private void OnTURNAllocate(TObject *Sender, const TsgcSocketConnection *aSocket,
const TsgcSTUN_Message *aMessage, const TsgcTURN_ResponseAllocation *aAllocation)
{
  DoLog("Relayed IP: " + aAllocation->RelayedIP + ". Relayed Port: " +
    IntToStr(aAllocation->RelayedPort));
}


private void OnSTUNResponseError(TObject *Sender, const TsgcSTUN_Message *aMessage,
const TsgcSTUN_ResponseError *aError)
{
  DoLog("Error: " + IntToStr(aError->Code) + " " + aError->Reason);
}
```

The lifetime can be updated to avoid expiration using the method **Refresh**. The Lifetime is the number of seconds to expire. If the value is zero the Allocation will be deleted.

```
oTURN->Refresh(600);
```

TURN Client | Create Permissions

When a new Allocation is created in a TURN server, this allocation cannot process any incoming packet from other peers if has no permissions. So, in order to allow other peers to communicate using a Relayed IP Address, first the TURN Client must create permissions for the IP Addresses that are allowed to exchange Data.

To Create a new Permission, just call the method **CreatePermission** and pass as a parameter the IP Address of the peer. If the Peer IP already exists on the TURN server, it will be refreshed, if not, it will be created. Permissions expire after 5 minutes unless are refreshed.

The TURN client, only allows you to call the method CreatePermission if exists an active allocation.

If the permission is created successfully, the event **OnTURNCreatePermission** is called.

```
oTURN->CreatePermission("80.147.23.157");


void OnTURNCreatePermission(TObject *Sender, const TsgcSocketConnection *aSocket,
  const TsgcSTUN_Message *aMessage, const TsgcTURN_ResponseCreatePermission *aCreatePermission)
{
  DoLog("#Create Permission: " + aCreatePermission->IPAddresses->Text);
}
```

TURN Client | Send Indication

TURN Protocol supports 2 mechanisms for sending and receiving data from peers, one of them is Send and Data mechanisms.

The TURN client can use the **SendIndication** method to send data to the server for relaying to a peer. The TURN client must ensure that there is a permission for the Peer IP Address where the Send Indication will be sent.

The responses to a SendIndication method, are received **OnTURNDataIndication** event.

```
oTURN->SendIndication("80.147.23.157", 5000, "random data");

void OnTURNDataIndication(TObject *Sender, const TsgcSocketConnection *aSocket,
  const TsgcSTUN_Message *aMessage, const TsgcTURN_ResponseDataIndication *aDataIndication)
{
  DoLog("#Data Indication: [" + aDataIndication->PeerIP + ":" + IntToStr(aDataIndication->PeerPort) + "] " +
    sgcGetStringFromBytes(aDataIndication->Data));
}
```

TURN Client | Channels

Channels provide a way for the TURN Client and Server to send application data using ChannelData messages, which have less overhead than Send and Data Indications.

Before using ChannelData messages to exchange data between peers, the TURN client must create a new channel, to do this, just call the method **ChannelBind** passing the Peer IP Address and Port as parameters.

If the TURN server can bind a new channel, the TURN client will receive a successful response **OnTURNChannelBind** event.

```
oTURN->ChannelBind("80.147.23.157", 5000);

void OnTURNChannelBind(TObject *Sender, const TsgcSocketConnection *aSocket,
  const TsgcSTUN_Message *aMessage, const TsgcTURN_ResponseChannelBind *aChannelBind)
{
  DoLog("#Channel Bind: " + IntToStr(aChannelBind->Channel));
}
```

A channel binding lasts for 10 minutes unless refreshed. To refresh a channel just call **ChannelBind** method again.

When the TURN client receives a new ChannelMessage, the event **OnTURNChannelData** is called.

```
void OnTURNChannelData(TObject *Sender, const TsgcSocketConnection *aSocket,
  const TsgcTURNChannelData *aChannelData)
{
  DoLog("#Channel Data: [" + IntToStr(aChannelData->ChannelID) + "] " +
    sgcGetStringFromBytes(aChannelData->Data));
}
```

TsgcTURNServer

TURN Server supports Long-Term Authentication, Allocation, Permissions, Channel Data and more.

Introduction

TsgcTURNServer is the server that implements the TURN protocol and allows you to process requests from TURN clients. The component inherits from TsgcSTUNServer, so all methods and properties are available on TsgcTURNServer.

TURN Server supports Long-Term Authentication, Allocation, Permissions, Channel Data and more.

Basic usage

Usually TURN servers run on UDP port 3478 and require Long-Term credentials, so in order to configure a TURN server, set the listening port (by default 3478) and start the server.

Configure the server

- Port: the listening Server port, example: 3478
- Define the Long-Term Credentials properties in TURNOptions.Authentication.LongTermCredentials
- Handle the OnSTUNRequestAuthorization to set the password when a TURN client sends a request to TURN Server.

Set the property **Active = True** to start the TURN server.

```
TsgcTURNServer oTURN = new TsgcTURNServer();
oTURN->Port = 3478;
oTURN->TURNOptions->Authentication->Enabled = true;
oTURN->TURNOptions->Authentication->LongTermCredentials->Enabled = true;
oTURN->TURNOptions->Authentication->LongTermCredentials->Realm = "esegece.com";
oTURN->Active = true;
void OnSTUNRequestAuthorization(TObject *Sender, const TsgcSTUN_Message *aRequest,
  const string aUsername, const string aRealm, ref string Password)
{
  if ((aUsername == "user") && (aRealm == "esegece.com"))
  {
    Password = "password";
  }
}
```

Reference

- Properties — 9 properties available on the component.
- Methods — 2 public methods.
- Events — 11 events you can handle.

Guides

- **Configurations**
  - Long-Term Credentials
- **Allocations**
  - Allocations

TURN Server | Long Term Credentials

Usually TURN Servers are configured WITH Authentication for TURN requests and without Authentication for STUN requests.

sgcWebSockets TURN Server supports Long-Term Credentials, so you can configure TsgcTURNServer to only allow requests with Long-Term credentials info.

To configure it, access the TURNOptions.Authorization property and enable it.
Then access to LongTermCredentials property and enabled it. By default, this type of authorization is already configured with a Realm string and with a default StaleNonce value of 10 minutes (= 600 seconds).

```
TsgcTURNServer oTURN = new TsgcTURNServer();
oTURN->STUNOptions->Authentication->Enabled = false;
oTURN->TURNOptions->Authentication->Enabled = true;
oTURN->TURNOptions->Authentication->LongTermCredentials->Enabled = true;
oTURN->TURNOptions->Authentication->LongTermCredentials->Realm = "sgcWebSockets";
oTURN->TURNOptions->Authentication->LongTermCredentials->StaleNonce = 600;
oTURN->Port = 3478;
oTURN->Active = true;


private void OnSTUNRequestAuthorization(TObject *Sender, const TsgcSTUN_Message *aRequest,
  const string aUsername, const string aRealm, ref string Password)
{
  if ((aUsername == "my-user") && (aRealm == "sgcWebSockets"))
  {
    Password = "my-password";
  }
}
```

TURN Server | Allocations

All TURN operations revolve around allocations and all TURN messages are associated with an Allocation. An allocation consists of:

- The relayed transport address
- The 5-Tuple: client's IP Address, client's IP port, server IP address, server port and transport protocol.
- The authentication information.
- The time-to-expiry for each relayed transport address.
- A list of permissions for each relayed transport address.
- A list of channels bindings for each relayed transport address.

When a TURN client sends an Allocate request, this TURN message is processed by server and tries to create a new Relayed Transport Address. By default, if there is any available UDP port, it will create a new Relayed Address, but you can use **OnTURNBeforeAllocate** event to reject a new Allocation request.

```
void OnTURNBeforeAllocate(TObject *Sender, const TsgcSocketConnection *aSocket,
  const string aIP, Word aPort, ref bool Reject)
{
  if (your own rules) == false
  {
        Reject = false;
  }
}
```

If the process continues, the server creates a new allocation and the event **OnTURNCreateAllocation** is called. This event provides information about the Allocation through the class TsgcTURNAllocationItem.

```
void OnTURNCreateAllocation(TObject *Sender, const TsgcSocketConnection *aSocket,
  const TsgcTURNAllocationItem *Allocation)
{
  DoLog("New Allocation: " + Allocation->RelayIP + ":" + IntToStr(Allocation->RelayPort));
}
```

When the Allocation expires or is deleted receiving a Refresh Request from client with a lifetime of zero, the event **OnTURNDeleteAllocation** event is fired.

```
void OnTURNDeleteAllocation(TObject *Sender, const TsgcSocketConnection *aSocket,
  const TsgcTURNAllocationItem *Allocation)
{
  DoLog("Allocation Deleted: " + Allocation->RelayIP + ":" + IntToStr(Allocation->RelayPort));
}
```

ICE

Interactive Connectivity Establishment (ICE) Protocol is used for NAT traversal. ICE uses a combination of methods including Session Traversal Utility for NAT (STUN) and Traversal Using Relay NAT (TURN). The presence of a Network Address Translator (NAT) presents problems for Voice over IP (VoIP) and WebRTC implementations.

Components

- **TsgcICEClient:** the client component that implements the ICE protocol and allows you to obtain, exchange, and verify candidates.

TsgcICEClient

ICEOptions.

Introduction

TsgcICEClient is the client that implements the ICE protocol and allows you to send allocation requests to TURN servers. The client requires the TsgcTURNClient and a TsgcWebSocketClient.

Configuration

The ICE client has the following properties

- **ICEOptions.CheckList.MaxCandidates:** is the max number of candidates that can handle the client, by default 100.
- **ICEOptions.GatherCandidates.STUN:** by default true, will obtain the candidates using the STUN protocol (reflexive address).
- **ICEOptions.GatherCandidates.TURN:** by default true, will obtain the candidates using the TURN protocol (relayed address).

The ICE client requires a **TURN client** to gather the candidates, so link a TsgcTURNClient to the TURN property of TsgcICEClient before calling the method **GatherCandidates**. You can obtain more information about how to configure the TURN client.

```
TsgcICEClient *oICE = new TsgcICEClient();
TsgcTURNClient *oTURN = new TsgcTURNClient();
oTURN->Host = "www.esegece.com";
oTURN->Port = 3478;
oTURN->TURNOptions->Authentication->Credentials = stauLongTermCredential;
oTURN->TURNOptions->Authentication->Username = "sgc";
oTURN->TURNOptions->Authentication->Password = "secret";
oICE->GatherCandidates();
```

Reference

- Properties — 3 properties available on the component.
- Methods — 3 public methods.
- Events — 6 events you can handle.

Guides

- **Candidates**
  - Gather Candidates
  - Pair Candidates

ICE | Gather Candidates

ICE starts gathering candidates, usually will obtain local IP Addresses, reflexive address using STUN protocol and relayed address using TURN protocol.

To start the gathering call the method **GatherCandidates**, this will start an internal timer where first will obtain the local IP addresses, then will connect to the STUN server to obtain the reflexive IP Address and finally will connect to TURN server to obtain the relayed IP Address.

Every time a new candidate is obtained, the event **OnICECandidate** will be called asynchronously, if there is any error while gathering the candidates, the event **OnICECandidateError** will be triggered.

```
TsgcICEClient *oICE = new TsgcICEClient();
TsgcTURNClient *oTURN = new TsgcTURNClient();
oTURN->Host = "www.esegece.com";
oTURN->Port = 3478;
oTURN->TURNOptions->Authentication->Credentials = stauLongTermCredential;
oTURN->TURNOptions->Authentication->Username = "sgc";
oTURN->TURNOptions->Authentication->Password = "secret";
oICE->GatherCandidates();

void OnICECandidate(TObject *Sender, const TsgcICE_Candidate *aCandidate)
{
  DoLog("[#Candidate] " + aCandidate->AsString);
}
```

ICE | Pair Candidates

Once the Candidates have been obtained (local and remote) and the SDP descriptions have been set, the ICE caller client can start to process all the pair candidates to find those that can exchange data. To start this process, call the method **ProcessCandidates**.

The method ProcessCandidates evaluate all pair candidates sending a STUN binding packet, if the STUN binding packet is received as an answer from the other peer, means the connection is possible between those 2 peers, so the pair is nominated.

When the pairing is **successful**, the event **OnICECandidatePairNominated** is triggered asynchronously. If the pairing has an **error** or **cannot connect after a timeout**, the event **OnICECandidatePairFailed** is triggered.

TsgcRTCPeerConnection

The TsgcRTCPeerConnection is a client component that allows you to connect peers using P2P through UDP.

Introduction

[*Currently this component is in development]

The TsgcRTCPeerConnection is a client component that allows you to connect peers using P2P through UDP. The flow can be broken into 4 steps:

- Signaling
- Connecting
- Securing
- Communicating

To implement those steps, the client makes use of the following protocols:

- **WebSocket:** this protocol is used for signaling, the clients exchange the Session Description Protocol and the local, public and Relayed IP addresses.
- **UDP:** this is the transport protocol, the client uses UDP to send/receive messages between peers.
- **DTLS:** similar to TLS, is an encryption specification that secures the message between peers, avoiding third-parties to read/write messages.
- **STUN:** protocol to obtain the public IP address.
- **TURN:** protocol to relay IP addresses when peers are behind NATs.
- **ICE:** protocol to find which IP Address and Ports are accessible between peers.

Signaling

When the client starts it has no idea who it is going to communicate with or what it is going to communicate about. Signaling uses the SDP (Session Description Protocol) which contains details like:

- IPs and Ports where the peer is reachable.
- Fingerprint's Certificate used to secure the communication.
- User and Password.
- ...

The Signaling makes use of the WebSocket protocol to exchange the data, it works through a subprotocol and it's implemented in the TsgcWSPServer_RTCPeerConnection component on server side.
The TsgcRTCPeerConnection already creates internally a websocket client with TsgcWSPClient_RTCPeerConnection attached.

To obtain the IPs and Ports, the client makes use of the STUN/TURN protocols to obtain this information. So a STUN/TURN server is required too.

Links:
- RTCPeerConnection WebSocket Server
- RTCPeerConnection WebSocket Client
- RTCPeerConnection STUN TURN
- RTCPeerConnection Signaling

Connecting

Once the 2 peers know the candidates and SDPs, the client uses another standard protocol called ICE.

ICE (Interactive Connection Establishment) allows the establishment of a connection between 2 peers. The peers can be in the same network or behind a NAT... ICE is a solution to establishing a direct connection without a central server. If the connection cannot be P2P, ICE will use TURN to relay the data using a TURN server.

Once ICE finds a valid candidate that can connect between the two peers, the next step is to encrypt the communication.

Links:
- RTCPeerConnection ICE

Securing

After the peers have connected, the communication must be secure. This is done using DTLS, which is a cryptographic protocol used to secure communication over UDP.

Once the DTLS handshake has been successfully processed, another protocol is used, SRTP (Secure Real-Time Transport Protocol), currently SRTP is not implemented.

Links:
- RTCPeerConnection DTLS

Communicating

Once the 2 peers are using a secure protocol, the communication is done using 2 protocols:

- **RTP:** Real Time Transport Protocol: used to exchange media encrypted with SRTP.
- **SCTP:** Stream Control Transmission Protocol, used to send and receive DataChannel messages encrypted with dTLS.

Currently these protocols are not implemented, but you can send/receive data using DTLS over UDP.

Links:
- RTCPeerConnection Data

Reference

- Properties — 2 properties available on the component.
- Methods — 3 public methods.
- Events — 14 events you can handle.

RTCPeerConnection | WebSocket Server

The TsgcRTCPeerConnection client requires a WebSocket Server for signaling. The client makes use of the WebSocket protocol to exchange the SDP of the peers and the candidates (IPs and ports), which will allow peers to communicate.

To configure a **WebSocket server** you can use any of the WebSocket servers available in the sgcWebSockets library and attach a **TsgcWSPServer_RTCPeerConnection** which is the sub-protocol used by the RTCPeerConnection.

```
TsgcWebSocketServer *oServer = new TsgcWebSocketServer();
TsgcWSPServer_RTCPeerConnection *oProtocol = new TsgcWSPServer_RTCPeerConnection();
oProtocol->Server = oServer;
oServer->Port = 8080;
oServer->Active = true;
```

Every time a new websocket client connects to the websocket server, the server will check if there is any other peer listening on the same channel and will forward the data accordingly.

RTCConnection | WebSocket Client

The RTCPeerConnection creates internally a websocket client with a custom sub-protocol to communicate to a websocket server. In the RTCOptions.WebSocket property you can find the values that define the websocket connection

- **Host:** dns or ip address of the server, example: 127.0.0.1 or www.esegece.com.
- **Port:** listening port of websocket server.
- **TLS:** enable it if the server is using a secure connection.
- **Channel:** the channel name used to exchange data between peers (both peers must have the same channel name and the max number of peers is 2).

RTCPeerConnection | STUN TURN

The TsgcRTCPeerConnection uses STUN/TURN protocol to obtain the public IP Address and the Relayed IP Address (if required). So you need a STUN/TURN Server to obtain this information. You can read more about STUN/TURN server from the following link: TURN Server.

Once you have your STUN/TURN server running, you can configure the TURN Server properties in the RTCOptions.ICE property of the TsgcRTCPeerConnection.

- **Host:** ip address or dns of the TURN server. Example: 127.0.0.1 or www.esegece.com.
- **Port:** usually is the default port 3478.
- **Username:** username if the TURN server is using Long-term credentials (the default).
- **Password:** password if the TURN server is using Long-term credentials (the default).

RTCPeerConnection | Signaling

Once the TsgcRTCPeerConnection has configured the RTCOptions property and the Servers (WebSocket and STUN/TURN) are running, the client can start the process of gathering candidates.

The client first connects to the websocket server, if the connection is successful, it sends the local SDP. Then tries to get the local and public IP Addresses, to get the public IP Addresses will send a binding request to the STUN server to obtain the public IP Address and the relayed IP Address of the TURN server. Every time it gets a new candidate, this info is passed to the websocket server which will be forwarded to the other peer.

When the RTCPeerConnection has the Local SDP, Remote SDP and the candidates it will start a process of checking every candidate pair to see if can connect between them. When a candidate pair successfully connects, it's a valid candidate pair and the Securing process continues the flow.

RTCPeerConnection | ICE

ICE (Interactive Connectivity Establishment) is the protocol used to connect 2 peers, it determines all the possible routes between the 2 peers and then ensures are connected. These routes are also known as Candidate Pairs, which is a pairing of local and remote transport address. These addresses can be the local IP Address, public IP Address or Relayed Transport Address. Each peer gathers all the addresses they want to use, exchanges them, and then attempt to connect.

Gathering Addresses

The following events can be called when gathering Addresses

OnRTCLocalCandidate

The event is called when a new local candidate has been found.

OnRTCRemoteCandidate

The event is called when the websocket server sends a remote candidate to this peer.

OnRTCLocalDescription

The event is called when the TsgcRTCPeerConnection requires the local SDP

OnRTCRemoteDescription

The event is called when the websocket server sends the remote SDP to this peer.

Connectivity Testing

When the peer sends binding requests to the other peer to test if can connect, the following events may be called

OnRTCCandidatePairNominated

When both peers can connect using this candidate pair, the event is called.

OnRTCCandidatePairFailed

When the peers cannot connect using this candidate pair, this event is called.

OnRTCConnect

This event is called when there is a valid candidate pair and DTLS is not enabled.
If DTLS is enabled, this event is called after a successful DTLS Handshake.

RTCPeerConnection | DTLS

Once there is a valid candidate pair (both peers can connect and exchange data between them), it's time to make the connection secure. DTLS is a cryptographic protocol that encrypts the data to avoid inspection or modification of the content of the data exchanged.

DTLS requires the openSSL libraries (from openSSL 1.1+), unless RTCOptions.DTLSOptions.IOHandler = iohNativeTLS
With **RTCOptions.DTLSOptions.IOHandler = iohNativeTLS** the DTLS handshake and the DTLS-SRTP keying run on the native DTLS 1.2 engine of the sgcCrypto pack, so WebRTC works without OpenSSL. The SRTP profiles (SRTP_AEAD_AES_128_GCM, SRTP_AES128_CM_SHA1_80, SRTP_AES128_CM_SHA1_32 and SRTP_AEAD_AES_256_GCM) are negotiated and the SRTP keys exported the same way as on the OpenSSL path. See SSL without OpenSSL.

The configuration of the DTLS can be found in the RTCOptions.DTLSOptions property of the TsgcRTCPeerConnection. To enable DTLS, set the property RTCOptions.DTLS to True. Find below the main properties:

RootCertFile: path to the trust store used to verify the other end, normally a file holding the certificate authority. This is the same meaning the property has on the TLS and QUIC transports. It is not the certificate of this endpoint, and a configuration that sets RootCertFile and KeyFile while leaving CertFile empty raises an error when the connection starts.
CertFile: path to the certificate file of this endpoint. The file can hold the certificate together with its intermediates, so a chain file works.
KeyFile: path to certificate key file.
VerifyCertificate: if the certificate chain of the other end must be verified, enable this property. WebRTC authenticates the other end by the fingerprint carried in the SDP and not by a certificate chain, so leaving this property disabled is the normal WebRTC configuration, and it is unaffected by the chain verification now enforced on the DTLS transports.
VerifyDepth: is an Integer property that represents the maximum number of links permitted when verification is performed, that is, the maximum chain depth allowed when the certificate of the other end is verified.
OpenSSL_Options: configuration of the openSSL libraries.
- **APIVersion:** allows defining which OpenSSL API will be used. only openSSL API 1.1+ supports DTLS.
  - **oslAPI_1_1:** uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).
  - **oslAPI_3_0:** uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).
- **LibPath:** here you can configure where are located the openSSL libraries
  - **oslpNone:** this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.
  - **oslpDefaultFolder:** sets automatically the openSSL path where the libraries should be located for all IDE personalities.
  - **oslpCustomFolder:** if this is the option selected, define the full path in the property LibPathCustom.
- **LibPathCustom:** when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.
- **UnixSymLinks:** enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
  - **oslsSymLinksDefault:** by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).
  - **oslsSymLinksLoadFirst:** Load SymLinks and do before trying to load the version libraries.
  - **oslsSymLinksLoad:** Load SymLinks after trying to load the version libraries.
  - **oslsSymLinksDontLoad:** don't load the SymLinks.

RTCPeerConnection | Data

Once the connection is successful, you can use the method **WriteData** to send some message to the other peer. The connection will make use of DTLS over UDP, and if possible the connection will be P2P (or using a relayed address when P2P is not possible).

Send Data

Use the method WriteData to send any data to the other peer. You can send a string or an array of bytes.

```
TsgcRTCPeerConnection *oRTCPeerConnection = new TsgcRTCPeerConnection();
...
oRTCPeerConnection->WriteData("Hello from sgcWebSockets!!!");
```

Receive Data

Every time the TsgcRTCPeerConnection receives any data from the other peer, the event OnRTCMessage will be called.

```
void OnRTCMessage(TObject *Sender, const TBytes aBytes)
{
  ShowMessage(TEncoding->UTF8->GetString(aBytes));
}
```

Datasnap

By default, DataSnap uses **TIdHttpWebBrokerBridge** as the server to handle HTTP requests. But you can expand the possibilities of your DataSnap application by replacing this server with another that supports more protocols and offers much better performance, taking advantage of the latest protocols like HTTP/2 which improves server performance, IOCP which allows you to handle many more connections, and more.

Servers

Server	Main Features	Description
TsgcWSHTTPWebBrokerBridgeServer	WebSocket Protocol HTTP 1.* Protocol XHR Protocol IOCP	Based on Indy library, supports WebSocket and HTTP protocols on the same port. IOCP can be enabled too.
TsgcWSHTTP2WebBrokerBridgeServer	WebSocket Protocol HTTP 1.* Protocol HTTP/2 Protocol XHR Protocol IOCP	Based on Indy library, supports WebSocket and HTTP/2 protocols on the same port. IOCP can be enabled too.
TsgcWSServer_HTTPAPI_WebBrokerBridge	WebSocket Protocol HTTP 1.* Protocol HTTP/2 Protocol XHR Protocol IOCP	Based on HTTP.SYS Microsoft HTTP API, supports WebSocket and HTTP/2 protocols on the same port. IOCP is used by default. Recommended for best performance.

TsgcWSHTTPWebBrokerBridgeServer

TsgcWSHTTPWebBrokerBridgeServer make use of **TIdHttpWebBrokerBridge** as server base and is useful if you want to use a single server for DataSnap, HTTP and WebSocket connections.

TsgcWSHTTPWebBrokerBridgeServer inherits from **TsgcWebSocketHTTPServer**, so you can refer to this server.

Follow next steps to replace TIdHttpWebBrokerBridge for TsgcWSHTTPWebBrokerBridgeServer :

1. Create a new instance of TsgcWSHTTPWebBrokerBridgeServer.

2. Replace all calls to TIdHttpWebBrokerBridge for TsgcWSHTTPWebBrokerBridgeServer.

3. To Handle WebSocket connections just refer to TsgcWebSocketHTTPServer.

Configuration

The **Datasnap** components are **only located in Source folder**, you won't find in the compiled folders because these objects are not included in sgcWebSockets package, so you must create in runtime.
Just add the required files to your project or set your path to the Source folder of sgcWebSockets package. Files required:

- sgcWebSocket_Server_WebBrokerBridge

If the project makes uses of IdHTTPWebBrokerBridge change to sgcIdHTTPWebBrokerBridge (this only applies for Enterprise Edition).

Events

```
FServer = new TsgcWSHTTPWebBrokerBridgeServer();
FServer->OnCommandRequest = OnCommandRequestEvent;
FServer->OnCommandGet = OnCommandGetevent;

void OnCommandRequestEvent(TIdContext *AThread, TIdHTTPRequestInfo *ARequestInfo,
TIdHTTPResponseInfo *AResponseInfo, ref bool aHandled)
{
  if (ARequestInfo->Document == "/test.html")
  {
    aHandled = true;
  }
}

void OnCommandGetevent(TIdContext *AThread, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo
*AResponseInfo)
{
  if (ARequestInfo->Document == "/test.html"
  {
    AResponseInfo->ResponseNo = 200;
    AResponseInfo->ContentText = "hello all";
  }
}
```

Load Balancer

If the server is behind the TsgcWebSocketLoadBalancerServer, you may have issues with CORS, to avoid these issues, use the following code

```
void __fastcall TWebModule1::WebModuleBeforeDispatch(TObject *Sender, TWebRequest *Request, TWebResponse *Respons
{
```

```
    Response->SetCustomHeader("Access-Control-Allow-Origin", "*");
    if (Trim(Request->GetFieldByName("Access-Control-Request-Headers")) != "")
    {
        Response->SetCustomHeader("Access-Control-Allow-Headers", Request->GetFieldByName("Access-Control-Request
        Handled = true;
    }
    if (FServerFunctionInvokerAction != nullptr)
    {
        FServerFunctionInvokerAction->Enabled = AllowServerFunctionInvoker;
    }
}
```

TsgcWSHTTP2WebBrokerBridgeServer

TsgcWSHTTP2WebBrokerBridgeServer use **TsgcWebSocketHTTPServer** with **HTTP/2** protocol enabled as server base and is useful if you want to use a single server for DataSnap, HTTP/2 and WebSocket connections.

TsgcWSHTTP2WebBrokerBridgeServer inherits from **TsgcWebSocketHTTPServer**, so you can refer to this server.

Follow next steps to replace TIdHttpWebBrokerBridge for TsgcWSHTTP2WebBrokerBridgeServer :

1. Create a new instance of TsgcWSHTTP2WebBrokerBridgeServer.

2. Replace all calls to TIdHttpWebBrokerBridge for TsgcWSHTTP2WebBrokerBridgeServer.

3. To Handle WebSocket connections just refer to TsgcWebSocketHTTPServer.

Configuration

The **Datasnap** components are **only located in Source folder**, you won't find in the compiled folders because these objects are not included in sgcWebSockets package, so you must create in runtime.
Just add the required files to your project or set your path to the Source folder of sgcWebSockets package. Files required:

- sgcWebSocket_Server_Base_WebBrokerBridge
- sgcWebSocket_Server_WebBrokerBridge_HTTP2

If the project makes uses of IdHTTPWebBrokerBridge change to sgcIdHTTPWebBrokerBridge (this only applies for Enterprise Edition).

Events

```
FServer = new TsgcWSHTTP2WebBrokerBridgeServer();
FServer->OnCommandRequest = OnCommandRequestEvent;
FServer->OnCommandGet = OnCommandGetevent;

void OnCommandRequestEvent(TIdContext *AThread, TIdHTTPRequestInfo *ARequestInfo,
TIdHTTPResponseInfo *AResponseInfo, ref bool aHandled)
{
  if (ARequestInfo->Document == "/test.html")
  {
    aHandled = true;
  }
}

void OnCommandGetevent(TIdContext *AThread, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo
*AResponseInfo)
{
  if (ARequestInfo->Document == "/test.html"
  {
    AResponseInfo->ResponseNo = 200;
    AResponseInfo->ContentText = "hello all";
  }
}
```

TsgcWSServer_HTTPAPI_WebBrokerBridge

TsgcWSServer_HTTPAPI_WebBrokerBridge use **TsgcWebSocketServer_HTTPAPI** with **HTTP/2** protocol enabled as server base and is useful if you want to use a single server for DataSnap, HTTP/2 and WebSocket connections.

TsgcWSServer_HTTPAPI_WebBrokerBridge inherits from **TsgcWebSocketServer_HTTPAPI**, so you can refer to this server.

Follow next steps to replace TIdHttpWebBrokerBridge for TsgcWSServer_HTTPAPI_WebBrokerBridge:

1. Create a new instance of TsgcWSServer_HTTPAPI_WebBrokerBridge.

2. Replace all calls to TIdHttpWebBrokerBridge for TsgcWSServer_HTTPAPI_WebBrokerBridge.

3. To Handle WebSocket connections just refer to TsgcWSServer_HTTPAPI_WebBrokerBridge.

Configuration

The **Datasnap** components are **only located in Source folder**, you won't find in the compiled folders because these objects are not included in sgcWebSockets package, so you must create in runtime.
Just add the required files to your project or set your path to the Source folder of sgcWebSockets package. Files required:

- sgcWebSocket_Server_Base_WebBrokerBridge
- sgcWebSocket_Server_HTTPAPI_WebBrokerBridge

If the project makes uses of IdHTTPWebBrokerBridge change to sgcIdHTTPWebBrokerBridge (this only applies for Enterprise Edition).

Events

```
TsgcWSServer_HTTPAPI_WebBrokerBridge FServer = new TsgcWSServer_HTTPAPI_WebBrokerBridge();
FServer->OnCommandRequest += OnCommandRequestEvent;
FServer->OnMessage += OnWebSocketMessage;

void OnCommandRequestEvent(TsgcWSConnection_HTTPAPI *aConnection,
  const THttpServerRequest *aRequestInfo, ref THttpServerResponse *aResponseInfo, ref bool aHandled)
{
  if (ARequestInfo->Document == "/test.html")
  {
    AResponseInfo->ResponseNo = 200;
    AResponseInfo->ContentText = "... body ...";
    aHandled = true;
  }
}
void OnWebSocketMessage(TsgcWSConnection *aConnection, const string aText)
{
  aConnection->WriteData(aText);
}
```

sgcHTML pages on DataSnap Servers

A DataSnap or WebBroker application hosted on one of the sgc bridge servers can serve a full **sgcHTML** web front-end on the same port as the REST API. The browser gets a page built with the sgcHTML components, complete with realtime WebSocket updates, while the /datasnap/* REST methods keep working exactly as before. There is no second server, no second port and no proxy.

Supported Bridge Servers

All three bridge servers are supported. Two engines make this possible, both hook the bridge server request pipeline and expose the same API:

Bridge server	Stack	Engine class
TsgcWSHTTPWebBrokerBridgeServer	Indy HTTP/1.1	TsgcHTMX_Engine_Server_WebBrokerBridge
TsgcWSHTTP2WebBrokerBridgeServer	HTTP/2 (and HTTP/1.1)	TsgcHTMX_Engine_Server_WebBrokerBridge (requires the SGC_HTTP2 define)
TsgcWSServer_HTTPAPI_WebBrokerBridge	HTTP.sys	TsgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge

Both engines descend from the sgcHTML engine family (TsgcHTMX_Engine_Server for the Indy and HTTP/2 bridges, TsgcHTMX_Engine_Server_HTTPAPI for the HTTP.sys bridge) and behave the same: pages, assets, router fragments and WebModule fall-through work identically on every bridge.

Wiring

Create the engine at runtime (there is no palette registration) and assign your bridge server:

Required units: **sgcHTMX_Engine_Server_WebBrokerBridge** for the Indy and HTTP/2 bridges (plus **sgcWebSocket_Server_WebBrokerBridge** for the server itself), or **sgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge** for the HTTP.sys bridge (plus **sgcWebSocket_Server_HTTPAPI_WebBrokerBridge**). The HTTP.sys engine is included in the sgcWebSockets Delphi packages and requires Delphi 10.1 Berlin or later, the other units are only located in the Source folder, add them to your project or set your search path accordingly. The engines belong to the HTML pack (SGC_HTMX define). The Indy and HTTP/2 bridges support Delphi 7 to Delphi 13.

Which Requests the Engine Claims

The engine intercepts only **GET** requests it can actually serve. Everything else falls through untouched to the WebBroker / DataSnap dispatcher:

Request	Handled by

GET / (the generated page)	sgcHTML engine
GET built-in js and css assets	sgcHTML engine
GET manifest.json and sw.js (when Template.PWAEnabled)	sgcHTML engine
GET paths registered in the Router	sgcHTML engine
/datasnap/* REST methods	DataSnap dispatcher
Any other request	WebBroker / DataSnap dispatcher

If you already assigned your own OnCommandRequest handler to the server, it keeps working: the engine chains it and your handler runs first. This applies on every bridge.

Realtime Push

The page opens a WebSocket connection back to the same port (Template.WSEndpoint, default '/htmx-ws'). Call **PushFragment** to push a live HTML fragment to a single browser, or **BroadcastFragment** to push it to all connected browsers. The interactive sgcHTML components (Grid, Chat, LogViewer, forms) work over the same WebSocket channel, so a DataSnap server can show a live dashboard with no browser-side polling.

Hosting without an sgc Server

If the application must run as ISAPI, Apache module or CGI, or you cannot replace the stock TIdHTTPWebBrokerBridge, use TsgcHTMX_Engine_Server_WebBroker instead. It plugs into any WebBroker application through IWebDispatch, but WebSocket push is not available in that hosting model, so the components degrade to standard HTMX HTTP round-trips and periodic updates use HTMX polling. See that topic for details.

Demo

Find a full example in the folder **Demos\40.DataSnap\Server_Indy_HTTP_HTML**: a DataSnap ReverseString server plus a realtime sgcHTML dashboard on a single port. The other variants in **Demos\40.DataSnap** host the same DataSnap server on the HTTP/2 and HTTP.sys bridges, and the wiring above applies to them unchanged.

See Also

- TsgcHTMX_Engine_Server_WebBrokerBridge
- TsgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge
- TsgcHTMX_Engine_Server_WebBroker
- TsgcWSHTTPWebBrokerBridgeServer
- TsgcWSHTTP2WebBrokerBridgeServer
- TsgcWSServer_HTTPAPI_WebBrokerBridge
- Datasnap

OpenAPI

OpenAPI 3.0

The **OpenAPI Specification**, previously known as the **Swagger Specification**, is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services. Previously part of the Swagger framework, it became a separate project in 2016, overseen by the OpenAPI Initiative, an open-source collaboration project of the Linux Foundation. Swagger and some other tools can generate code, documentation, and test cases given an interface file.

Applications implemented based on OpenAPI interface files can automatically generate documentation of methods, parameters and models. This helps keep the documentation, client libraries, and source code in sync.

Pascal Parser

sgcOpenAPI Generator allows generation of API client libraries (SDK generation) automatically given an OpenAPI Spec, the following OpenAPI specifications are supported:

- **OpenAPI 3.***
- **Swagger 2.*** (automatically converted from 2.0 to 3.0)

sgcOpenAPI allows you to generate **automatically** the client API interface in **Native Pascal Language** given a JSON/YAML OpenAPI or Swagger. Currently supports from Delphi 7 to latest Delphi version.

sgcOpenAPI Generator allows you to create a documentation file from an OpenAPI / Swagger specification.

Read more about OpenAPI Parser Pascal.

OpenAPI Client

The Client Interface generated contains all the functions/methods defined in the OpenAPI specification. The constants and enumerations are created too.

The following **Authentication** methods are supported:

- Basic Authentication
- OAuth2 Code (interactive)
- OAuth2 Credentials (non-interactive)
- JWT

Read more about OpenAPI Client.

OpenAPI Server

You can also host an OpenAPI 3.0 service in Delphi using the built-in OpenAPI server. The server auto-routes requests against your spec, serves the spec at `/openapi.json` and a Swagger UI at `/docs`, and supports CORS and request/body validation out of the box.
Two ways to define the API:

- Spec-First – load an existing OpenAPI 3.0 JSON/YAML file at runtime.
- Code-First – annotate a Delphi class with RTTI attributes; the spec is generated automatically (requires Delphi XE7+).

Read more about OpenAPI Server.

APIs

The following APIs have been compiled and are supported:

- Amazon AWS
- Google Cloud APIs
- Microsoft Azure
- Other APIs

OpenAPI | Parser Pascal

The **sgcOpenAPI Parser** reads the **OpenAPI 3.0 Specification in JSON or YAML** Format and creates automatically a **Delphi Client in Native Pascal Code**.

The **sgcOpenAPI Parser** is compatible with the following specifications:

- **OpenAPI 3.***
- **Swagger 2.*** (converted from 2.0 to 3.0 locally, nothing is uploaded)

The specification file can be in **JSON** or in **YAML** format. The parser allows you to import specifications split into **multiple schemas**, if you provide an URL it will try to download the specifications and if you provide a filename it will try to load the external schemas from your local drive.

A **YAML** document is read locally, always. There is no switch to enable, no service is contacted and nothing leaves your machine, so a specification covered by an NDA or a machine with no internet access is handled like any other. The **.yaml** and the **.yml** extensions are both recognised, and a file with no extension or with a **.txt** extension is detected by its content. A **Swagger 2.0** document is converted to OpenAPI 3 locally too. The **-r** switch, see the command line topic, is only a fallback for a document the parser cannot turn into a usable OpenAPI 3 specification on its own, and it is the only case in which the file is uploaded to **converter.swagger.io**.

A specification which is not valid **UTF-8** is read as **Windows-1252** and a warning is recorded, instead of failing. A JSON document should be UTF-8 (RFC 8259), so in a file that has been hand edited on Windows the characters outside the ASCII range may not be the expected ones.

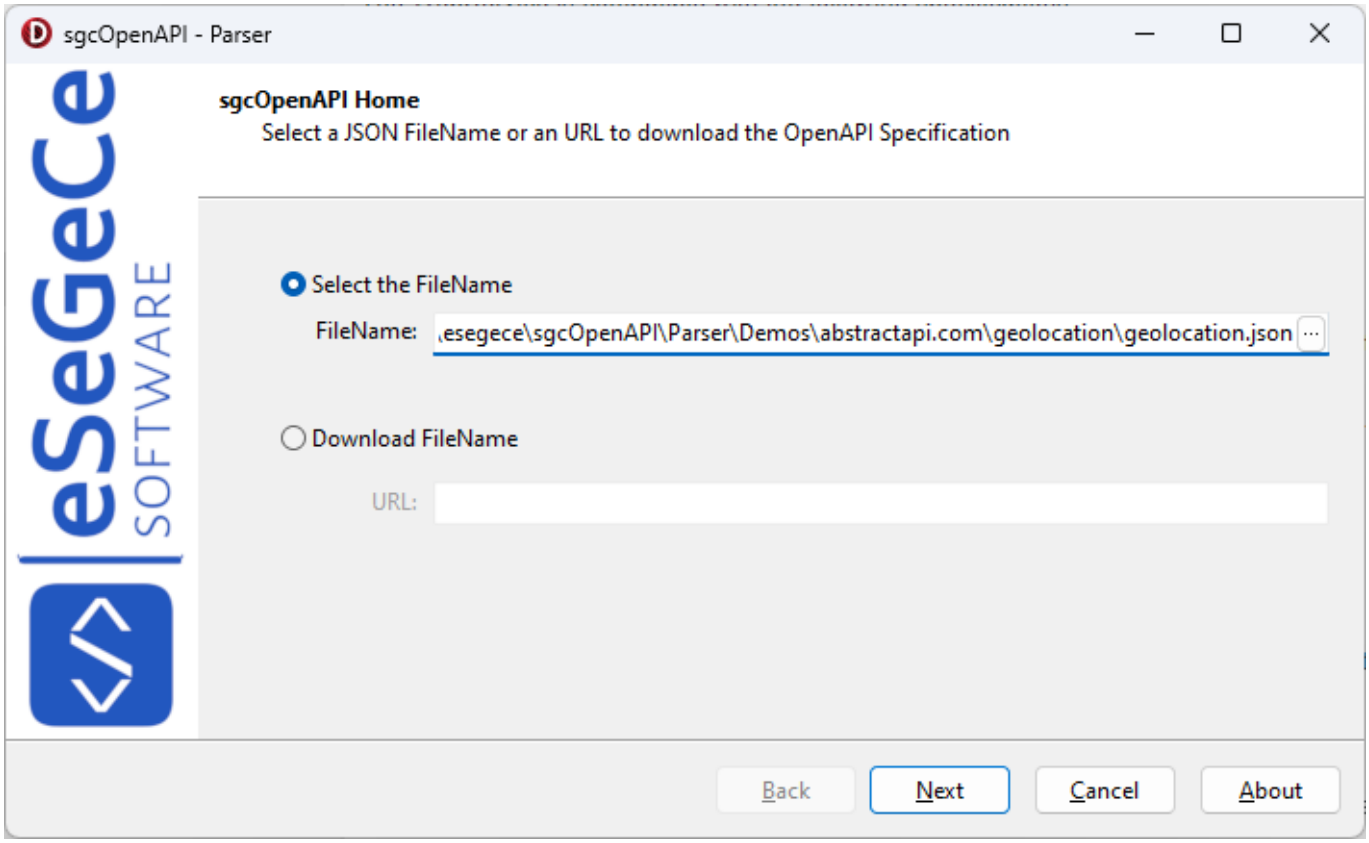
The name of the **generated unit follows the name of the output file**, because a unit only compiles when its name is the base name of its own file.

The OpenAPI Parser executable can handle command line parameters or use as a library.

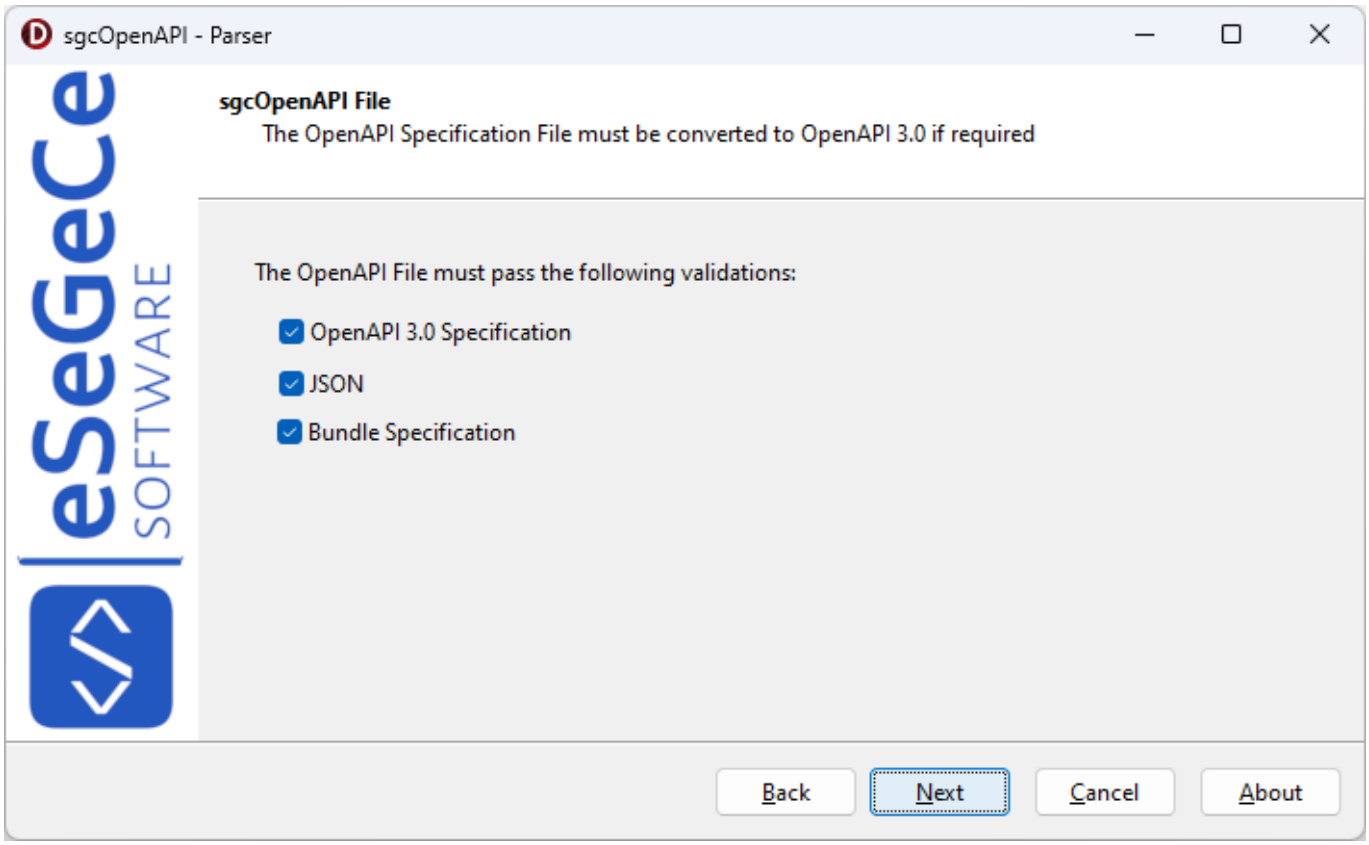
Importing OpenAPI Specification

The first step is to import the openAPI specification. Once you've the openAPI 3.0 specification in JSON format, you can generate the required Delphi files using our **OpenAPI WebService**. Follow the steps below:

- **Execute the sgcOpenAPI Parser** and select the specification to import or the URL to download.

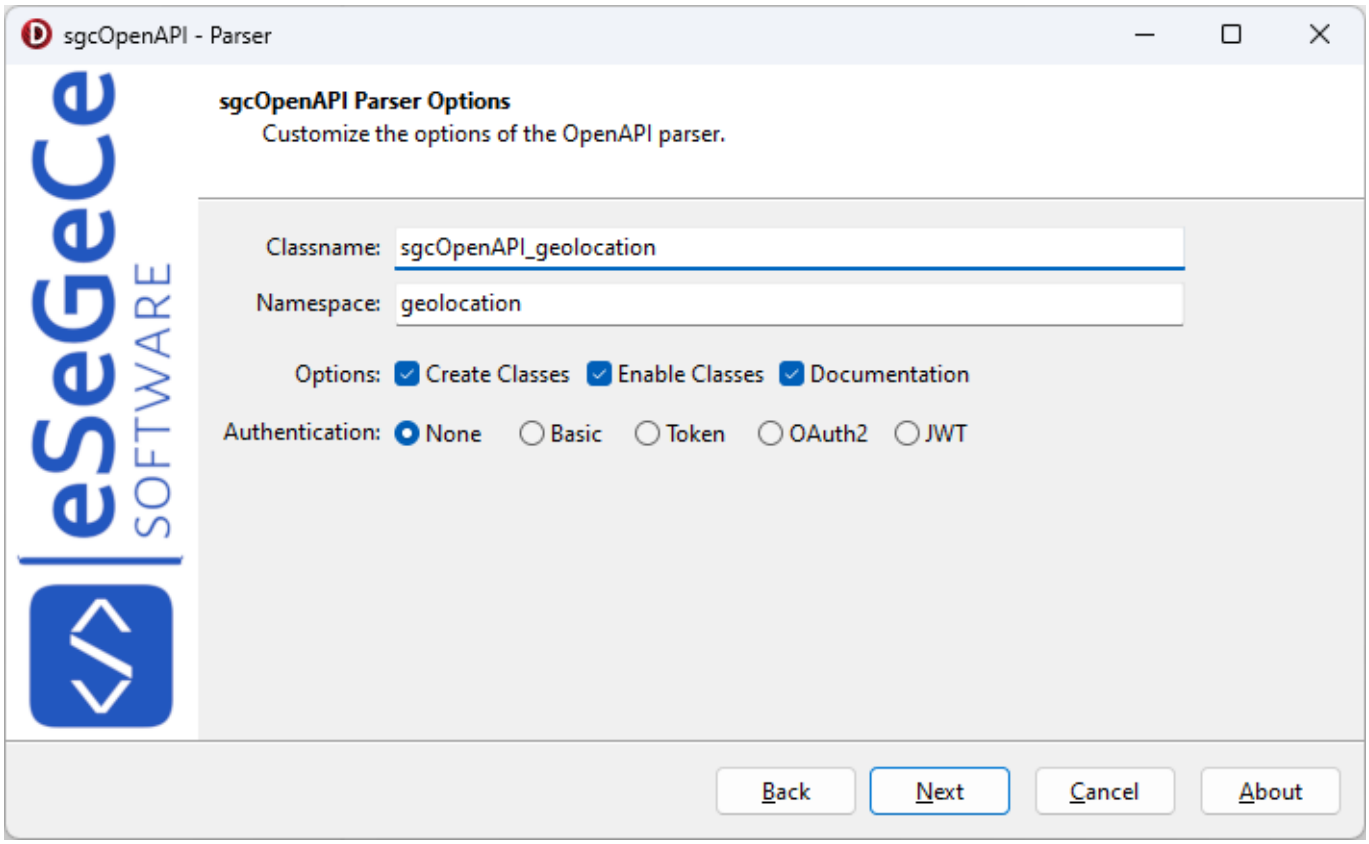


- The **specification is verified** and if it's compatible with the Parser you can continue to the next steps. A YAML document is read locally and a specification which makes use of an old version like swagger 2.0 is converted to OpenAPI 3 locally, in both cases nothing is uploaded. If the specification is split into multiple schemas, the parser will try to bundle a single specification. If there is any problem while converting the file, an error message will appear.

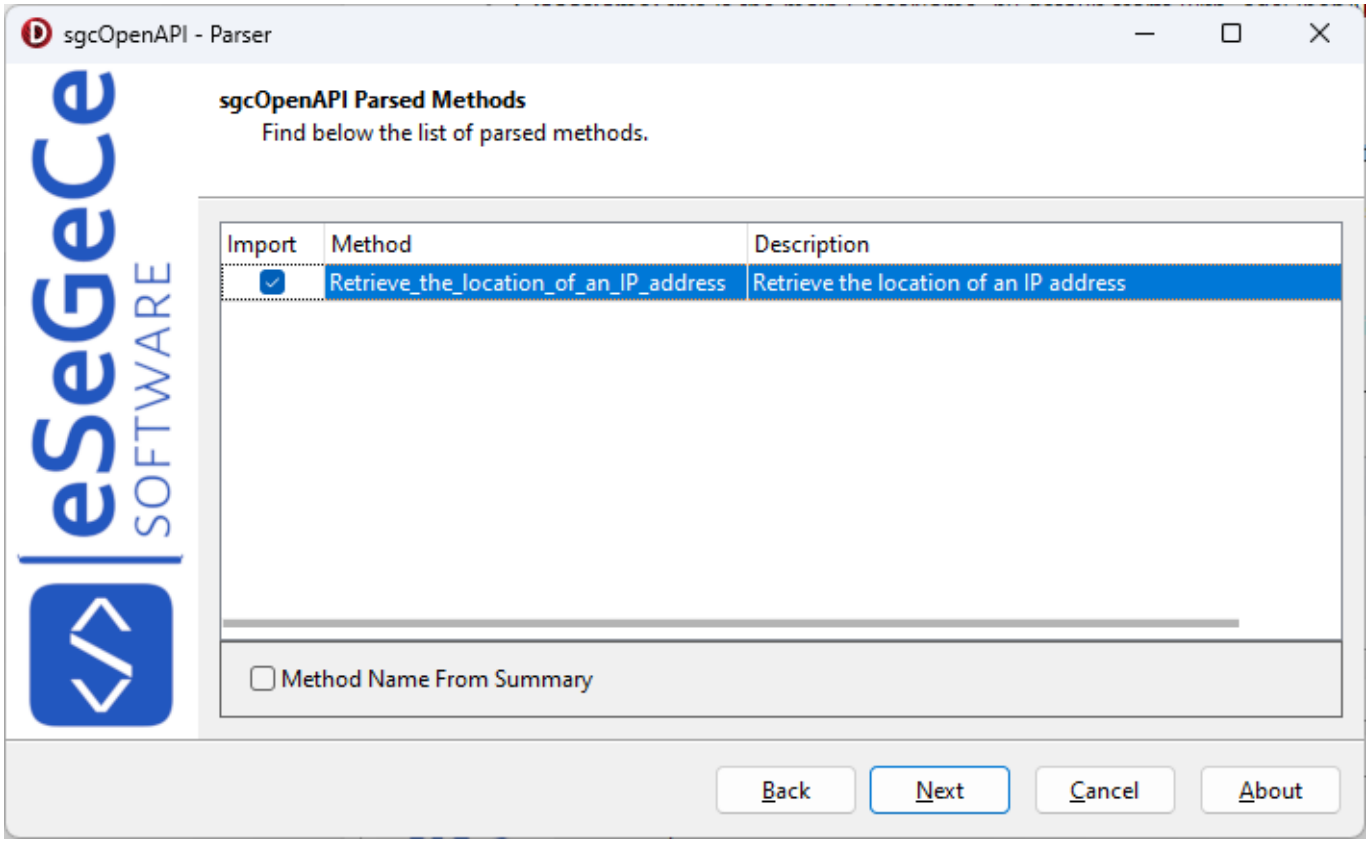


- Now you can **customize the following options** before parsing the document
  - **ClassName:** this is the main ClassName, by default starts with "sgcOpenAPI_" and adds the name of the specification filename.

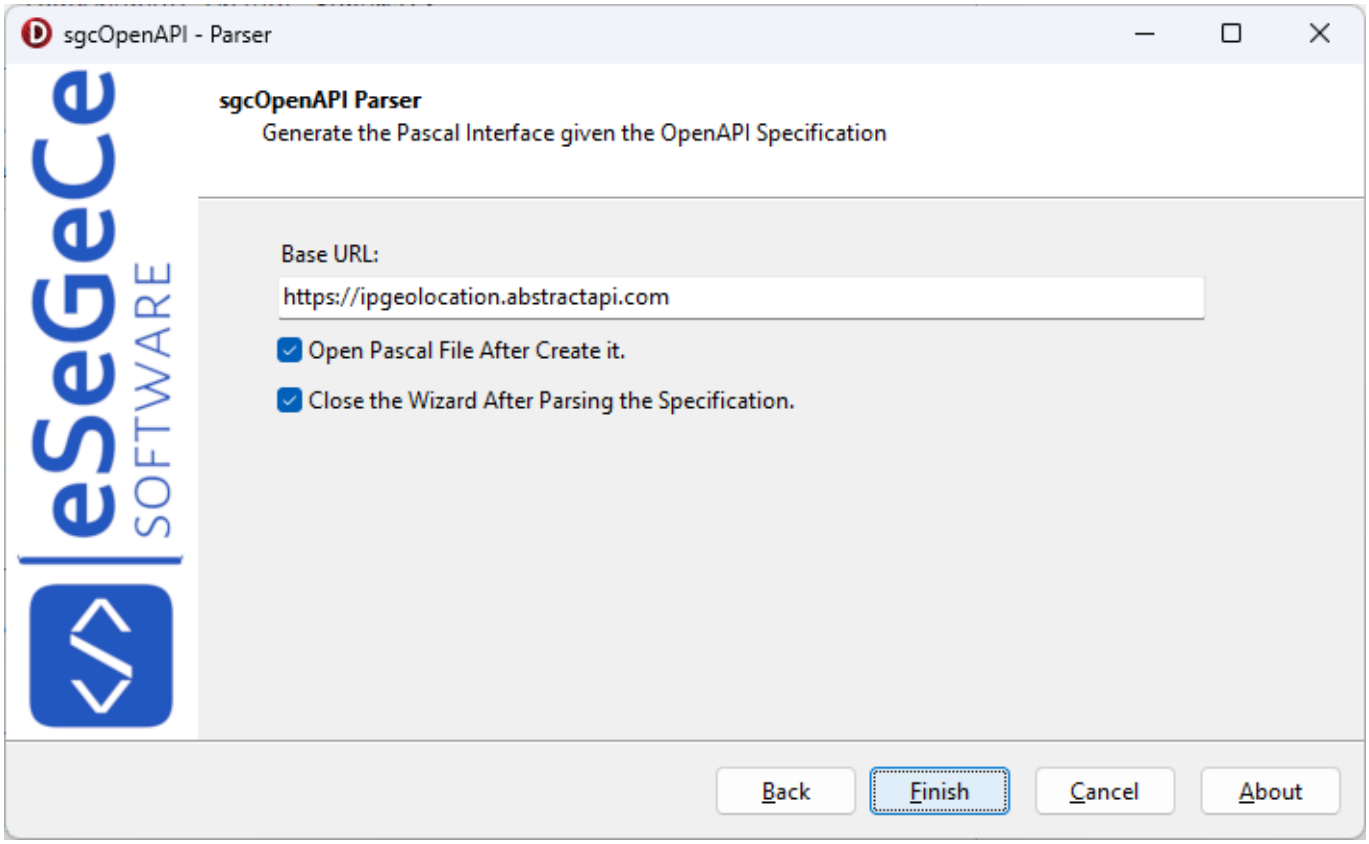
- **Namespace:** this is the name of the pascal file generated, by default is the same name of the specification file with the extension ".pas".
- **Create Classes:** if checked, it will create the classes from the specification file. Example: if a request requires to send a JSON object, and this JSON object is specified, the parser will create a class with the fields of the JSON object.
- **Enable Classes:** if checked, enables the Classes Generated from the specification file to use with JSON objects (Requires Rad Studio XE7+).
- **Documentation:** if checked, the parser will add comments to the fields, classes and methods.
- **Authentication:** select any authentication if exists.
  - **None:** the API doesn't make use of any authentication method.
  - **Basic:** the API makes use of BASIC authentication.
  - **Token:** is required to send a Token as an HTTP Header. This token is obtained from any other external method.
  - **OAuth2:** the request will use OAuth2 to authenticate the HTTP Requests.
  - **JWT:** the request will use JWT to authenticate the HTTP Requests.



- Now a **grid with a list of methods parsed** will be shown as information. By default, the parser takes the OperationId as a name for the methods created, but here you can use the summary as method name if the OperationId is not defined or the value is not valid.



- Finally, verify the **Base URL** has the correct value and here you can customize if the generated file will be opened automatically after created and if the wizard will be closed after generate the file.



- Press **Finish to parse the specification** and generate the pascal file.

Server Code Generation

Besides the Delphi client, the parser can generate a **server** skeleton from the specification (contract-first). Enable the **GenerateServer** option, or pass the **-server** flag on the command line, and the parser emits an attribute-decorated service class that mirrors the spec: one method per operation, decorated with **sgcServiceContract**, the verb attributes (**sgcHttpGet**, **sgcHttpPost** and so on), **sgcRoute**, and the parameter attributes **sgcFromPath** / **sgcFromQuery** / **sgcFromBody** / **sgcFromHeader**. The generated methods raise a "Not implemented" exception, so you only need to fill in the business logic. The shared model classes (DTOs) are generated as well.

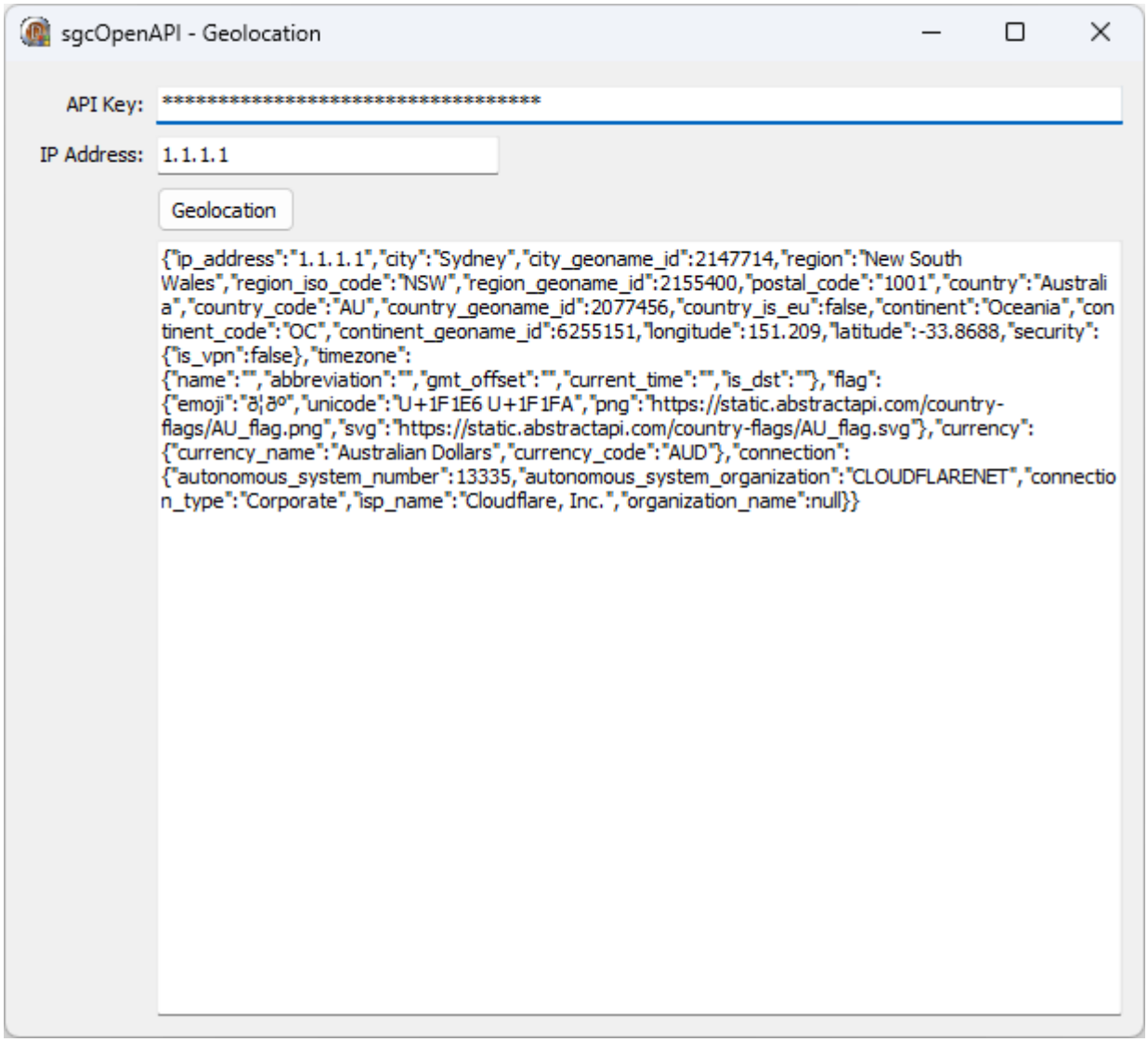
Example

I will use a simple openAPI specification used by abstractapi.com to retrieve the location of an IP Address.

Before test the demo, you must create a free account in abstractapi.com to get an API Key.

https://app.abstractapi.com/users/signup

Install the sgcOpenAPI Parser Setup, open the sgcOpenAPI.exe and import the OpenAPI specification that is located in the folder "Demos\abstractapi.com\geolocation". Once imported and stored in the same folder of the demo with the name "geolocation.pas", open the demo project, compile and execute it. Fill the API key obtained from the Abstractapi website and press Geolocation.



OpenAPI | Additional Properties

A dictionary (also known as a map, hashmap or associative array) is a set of key/value pairs. OpenAPI lets you define dictionaries where the keys are strings. To define a dictionary, use type: object and use the additionalProperties keyword to specify the type of values in key/value pairs. For example, a string-to-string dictionary like this:

```
{
  "en": "English",
  "fr": "French"
}
```

The OpenAPI Parser makes use of the JSON classes from Embarcadero, and converting a TDictionary to JSON, instead of creating a key/value pairs, it creates all the internal objects of the TDictionary, so the output is incorrect. The same applies when trying to convert a json string to a TDictionary object.

Convert AdditionalProperties to JSON

The OpenAPI Parser creates the AdditionalProperties classes as type of TsgcAdditionalProperties, so if you must convert a class to json string, you must create a new class that inherits from TsgcAdditionalProperties, create all the fields you need and then assign to the property class.

Example: given a class with 2 properties, one is an Additional properties and the json output must be

```
{"Property1":"value1","AdditionalProperties":{"key1":"value1", "key2":"value"}
TMyClass = class(TsgcOpenAPIClass)
private
  FProperty1: string;
  FAdditionalProperties: TsgcAdditionalProperties;
public
  property Property1: string read FProperty1 write FProperty1;
  property AdditionalProperties: TsgcAdditionalProperties read FAdditionalProperties write FAdditionalProperties;
end;
```

Create a new class that inherits from TsgcAdditionalProperties and create 2 properties

```
TCustomAdditionalProperties = class(TsgcAdditionalProperties)
private
  FKey1: string;
  FKey2: string;
public
  property Key1: string read FKey1 write FKey1;
  property Key2: string read FKey2 write FKey2;
end;
```

Finally Assign this class to the AdditionalProperties property:

```
TMyClass *oClass = new TMyClass();
oClass->Property1 = "value1";
TCustomAdditionalProperties *oAdditionalProperties = new TCustomAdditionalProperties();
oAdditionalProperties->Key1 = "value1";
oAdditionalProperties->Key2 = "value2";
oClass->AdditionalProperties = oAdditionalProperties;
```

Convert JSON to AdditionalProperties

Due to the limitations of the JSON classes, the OpenAPI Generator creates an additional method to load the AdditionalProperties in the TsgcAdditionalProperties.Dictionary property after the JSON string is parsed. So to access the content of the AdditionalProperties, just access to the TsgcAdditionalProperties.Dictionary property.

OpenAPI | Command Line

The **sgcOpenAPI Parser** includes a command line interface that converts an OpenAPI specification into a Delphi (Pascal) client file. The parser validates and normalizes the specification (OpenAPI 3 JSON + bundled schemas) before generating output.

Usage

Basic syntax

```
sgcOpenAPI.exe -i "c:\openapi.json" -o "c:\openapi.pas"
```

Switch Syntax

The value of a switch is read from the next parameter, `-i "c:\openapi.json"`, or from the text appended after a colon, `-i:"c:\openapi.json"`. A switch that takes no value is written on its own, `-nc`.

The form where the value is appended to the switch name without a separator is not accepted any more. `/o"c:\openapi.pas"` has to be written `-o "c:\openapi.pas"`, and `-x"GET /pets"` has to be written `-x "GET /pets"`. The switch name is compared for equality, a name that is not one of the switches below is reported as an error instead of being ignored.

Parameters

All parameters are case-insensitive. Short and long forms are equivalent.

- **-i -input** : **[mandatory]** Path to the OpenAPI specification or an URL to download. The document can be **OpenAPI 3** in JSON or in **YAML** format. A YAML specification is read locally, nothing is uploaded anywhere, and a **Swagger 2.0** document is converted to OpenAPI 3 locally too.
- **-o -output** : **[mandatory]** Output Pascal file to generate.
- **-s -server** : **[optional]** Generate a server skeleton (contract-first) instead of a client. The output is an attribute-decorated service class you implement.
- **-config -c** : **[optional]** Load all settings from a JSON configuration file. The keys read from the file are **input**, **output**, **classes**, **json**, **documentation**, **method**, **auth**, **url**, **log**, **server**, **exclude**, **prune** and **remote**. Values passed explicitly on the command line override the values read from the config file, but a boolean enabled in the configuration file cannot be disabled from the command line, a switch only turns an option on.
- **-nc -nclasses** : **[optional]** Do not generate Delphi classes for schemas (only interfaces / client methods).
- **-nj -njson** : **[optional]** Do not enable JSON serialization for generated classes.
- **-nd -ndoc** : **[optional]** Do not generate documentation comments in the output.
- **-m -method** : **[optional]** Method naming strategy for generated methods:
  - **0** : Use **OperationId** (default).
  - **1** : Use **Summary**.
  - **2** : Use **Endpoint** path.
- **-a -auth** : **[optional]** Authentication mode to enable in the generated client:
  - **0** : None.
  - **1** : Basic.
  - **2** : Token.
  - **3** : OAuth2.
  - **4** : JWT.
- **-u -url** : **[optional]** Override the base URL detected in the OpenAPI file.
- **-x -exclude** : **[optional]** Leave endpoints out of the generated code. Accepts a comma separated list of **VERB endpoint** entries, for example `-x "GET /pets,DELETE /pets/{petId}"`, or the name of a file with one entry per line. The switch can be repeated. It is also settable with the **exclude** key of the configuration file, when it is passed on the command line it replaces the list read from the file.

- **-p -prune** : **[optional]** Generate only the classes the selected endpoints actually use. It is also settable with the **prune** key of the configuration file.
- **-r -remote** : **[optional]** Allow the parser to send the specification to the public converter at **converter.swagger.io**. It is disabled by default and it is not needed for the usual formats, a YAML specification is read locally and a Swagger 2.0 document is converted to OpenAPI 3 locally, both without uploading anything. The switch is a fallback for a document the parser cannot turn into a usable OpenAPI 3 specification on its own. With it the complete specification, every endpoint, parameter, schema and internal hostname, is uploaded to a third party service which is not controlled by eSeGeCe, so enable it only for documents you can share. Without it nothing is sent and the file never leaves your machine. It is also settable with the **remote** key of the configuration file.
- **-l -log** : **[optional]** Show the progress messages while parsing. It does not affect the error messages, which are written to the standard error output and are always shown.
- **-user** : **[optional]** License user name (only required when the license has not been activated).
- **-password** : **[optional]** License password (only required when the license has not been activated).
- **-h -help** : Show command line help.

Examples

1) Generate Delphi client from a local file

```
sgcOpenAPI.exe -i "c:\specs\openapi.json" -o "c:\clients\orders.pas"
```

2) Generate client from URL, disable classes and documentation

```
sgcOpenAPI.exe -i "https://api.example.com/openapi.json" -o "c:\clients\api.pas" -nc
-nd
```

3) Use Summary as method names and enable OAuth2

```
sgcOpenAPI.exe -i "c:\specs\openapi.json" -o "c:\clients\auth.pas" -m 1 -a 3
```

4) Override Base URL and show log output

```
sgcOpenAPI.exe -i "c:\specs\openapi.json" -o "c:\clients\override.pas" -u "https://
api.example.com/v2" -l
```

5) Generate a server skeleton (contract-first)

```
sgcOpenAPI.exe -i "c:\specs\openapi.json" -o "c:\server\api.pas" -server
```

6) Load all settings from a JSON configuration file

```
sgcOpenAPI.exe -config "c:\specs\gen.json"
```

7) Exclude two endpoints and generate only the classes used by the rest

```
sgcOpenAPI.exe -i "c:\specs\openapi.json" -o "c:\clients\pets.pas" -x "GET /
pets,DELETE /pets/{petId}" -p
```

8) Generate a client from a YAML specification, it is read locally and nothing is uploaded

```
sgcOpenAPI.exe -i "c:\specs\openapi.yaml" -o "c:\clients\api.pas"
```

Exit Codes

The parser sets the exit code of the process, so a build script can detect a failure. Earlier versions always returned 0.

- **0** : success.
- **1** : error.
- **2** : invalid license.
- **3** : invalid switch.
- **4** : invalid configuration file.
- **5** : invalid input file.
- **6** : invalid output file.
- **7** : the file cannot be converted.

Notes

- A specification split into multiple schemas is bundled automatically before the code is generated.
- A **YAML** specification is read locally, always, with no switch to enable and nothing uploaded anywhere. The **.yaml** and the **.yml** extensions are both recognised, and a file with no extension or with a **.txt** extension is detected by its content. A **Swagger 2.0** document is converted to OpenAPI 3 locally too, so neither of them needs **-r**.
- When using URLs, the parser downloads the specification before validating and generating the output.
- The error messages are written to the standard error output and are always shown, the **-l** switch only controls the progress messages.

OpenAPI | Library

The OpenAPI Parser can be used as a shared library (DLL) so you can generate Pascal clients from your own applications without invoking the command line tool. The library exports a small set of functions that wrap the same parser features available in the CLI.

Library functions

- **CreateOpenAPIPascalFile** - creates a Pascal client file from an OpenAPI specification using default options.
- **CreateOpenAPIPascalFileEx** - creates a Pascal client file with full control over parser options.
- **GetOpenAPILastErrors** - returns the last error message produced by the parser.

CreateOpenAPIPascalFile

Generates a Pascal client file using the default parser options (generate classes, JSON support, documentation, method names from **OperationId**, no authentication, no base URL override).

```
typedef bool (__stdcall *TCreateOpenAPIPascalFile)(const char* aInputFile, const char* aOutputFile);
```

Parameters

- **aInputFile:** full path (or URL) of the OpenAPI specification to parse. A **YAML** specification is read locally and a **Swagger 2.0** document is converted to OpenAPI 3 locally, so both are normalized to OpenAPI 3 JSON before the output is generated and nothing is uploaded anywhere.
- **aOutputFile:** full path of the Pascal unit to create.
-

Return value

Returns **True** if the file is generated successfully, otherwise **False**. If it fails, call **GetOpenAPILastErrors** to retrieve the error message.

Example

CreateOpenAPIPascalFileEx

Generates a Pascal client file with explicit control over parser options.

```
typedef bool (__stdcall *TCreateOpenAPIPascalFileEx)(
  const char* aInputFile, const char* aOutputFile,
  bool aGenerateClasses, bool aGenerateJSON, bool aGenerateDocumentation,
  int aMethodsName, int aAuthentication, const char* aBaseURL, bool aEnableLog);
```

Parameters

- **aInputFile:** full path (or URL) of the OpenAPI specification to parse.
- **aOutputFile:** full path of the Pascal unit to create.
- **aGenerateClasses:** when **True**, the generator creates the Pascal models (classes) defined in the schema. When **False**, only the client interface is generated.
- **aGenerateJSON:** when **True**, JSON serialization helpers are generated in the classes. When **False**, JSON support is omitted.
- **aGenerateDocumentation:** when **True**, a documentation file for the OpenAPI specification is generated.

- **aMethodsName:** controls how method names are generated:
  - **0** = OperationId (default).
  - **1** = Summary.
  - **2** = Endpoint.
- **aAuthentication:** adds authentication helpers to the client:
  - **0** = None.
  - **1** = Basic.
  - **2** = Token.
  - **3** = OAuth2.
  - **4** = JWT.
- **aBaseURL:** override the base URL from the OpenAPI specification. Pass an empty string to keep the original value.
- **aEnableLog:** when **True**, the parser writes progress and validation messages that can help with troubleshooting.

Return value
Returns **True** if the file is generated successfully, otherwise **False**. If it fails, call **GetOpenAPILastErrors** to retrieve the error message.

Example

GetOpenAPILastErrors

Returns a message describing the last error (validation, conversion, or parsing error) produced by the parser. Call this function after **CreateOpenAPIPascalFile** or **CreateOpenAPIPascalFileEx** returns **False**.

```
typedef const char* (__stdcall *TGetOpenAPILastErrors)();
```

Return value
Returns the count of errors

Example

OpenAPI | API

The sgcOpenAPI Parser allows to use a custom dll to get more detailed info of the parsed pascal file, just create a new dll with the method **sgcOpenAPI_CreateSection** and copy the library in the same folder where is the parser. Every time a new file section is created the api the method CreateSection will be called and you can modify the content if needed. Find below a simple delphi example:

```
enum TsgcOpenAPIPascalSectionType {
  soapistHeaderStart, soapistHeaderEnums,
  soapistHeaderClasses, soapistHeaderMainClass, soapistHeaderClassesStructure,
  soapistHeaderMainInstance, soapistHeaderEnd, soapistImplementationStart,
  soapistImplementationClasses, soapistImplementationEnd
};

extern "C" __declspec(dllexport) void __stdcall sgcOpenAPI_CreateSection(
  int aType, wchar_t* &aValue)
{
  switch ((TsgcOpenAPIPascalSectionType)aType) {
    case soapistHeaderStart: break;
    case soapistHeaderEnums: break;
    case soapistHeaderClasses: break;
    case soapistHeaderMainClass: break;
    case soapistHeaderClassesStructure: break;
    case soapistHeaderMainInstance: break;
    case soapistHeaderEnd: break;
    case soapistImplementationStart: break;
    case soapistImplementationClasses: break;
    case soapistImplementationEnd: break;
  }
}
```

OpenAPI | Client

TsgcOpenAPI_Client is a non-visual component that encapsulates the main methods and properties to make HTTP requests from an OpenAPI specification.

Every OpenAPI interface created with sgcOpenAPI Parser has **2 methods**

1. **GetOpenAPIClient:** it's a singleton function that returns an instance of the main class, if not exists, it creates automatically.
2. **FreeOpenAPIClient:** frees the main class if it's created.

Example
Use the Abstractapi to retrieve the localization of an IP Address.

```
GetOpenAPIClient->Retrieve_the_location_of_an_IP_address("your api", "80.258.15.2");
```

Authentication

- **Basic:** uses a username/password as an HTTP header to authenticate.
  - **UserName:** name of the user.
  - **Password:** secret.
- **OAuth2:** authenticates using OAuth2, supports 2 types of Authorization:
  - **auth2Code:** It's used to perform authentication and authorization in the majority of application types, including single page applications, web applications, and natively installed applications. The flow enables apps to securely acquire access_tokens that can be used to access resources secured, as well as refresh tokens to get additional access_tokens, and ID tokens for the signed in user.
  - **auth2ClientCredentials:** This type of grant is commonly used for server-to-server interactions that must run in the background, without immediate interaction with a user. These types of applications are often referred to as daemons or service accounts.
  - Read more about OAuth2.
- **JWT:** authenticates using JWT. Read more about JWT.
- **Token:** you can pass a bearer token or any other custom token value.

TLSOptions

Allows to configure how connect to secure SSL/TLS servers using HTTP/1 protocol

The generated clients **verify the certificate of the server by default**, earlier versions did not. The trust comes from the OpenSSL default verify paths, so on a machine where no CA store is configured, set **TLSOptions.RootCertFile** to the file with the trusted root certificates. To connect to a self-signed or test endpoint, set **TLSOptions.VerifyCertificate := False**.

ALPNProtocols: list of the ALPN protocols which will be sent to server.
RootCertFile: path to root certificate file.
CertFile: path to certificate file.
KeyFile: path to certificate key file.
Password: if certificate is secured with a password, set here.
VerifyCertificate: if certificate must be verified, enable this property. The OpenAPI client enables it by default, set it to False to accept a self-signed certificate.
VerifyDepth: is an Integer property that represents the maximum number of links permitted when verification is performed for the X.509 certificate.
Version: by default uses TLS 1.0, if server requires a higher TLS version, here can be selected.
IOHandler: select which library you will use to connection using TLS.
iohOpenSSL: uses OpenSSL library and is the default for Indy components. Requires to deploy openssl libraries for win32/win64.

iohSChannel: uses Secure Channel which is a security protocol implemented by Microsoft for Windows, doesn't require to deploy openssl libraries. Only works in Windows 32/64 bits.

OpenSSL_Options: configuration of the openSSL libraries.

APIVersion: allows defining which OpenSSL API will be used.

osIAPI_1_0: uses API 1.0 OpenSSL, it's latest supported by Indy

osIAPI_1_1: uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).

osIAPI_3_0: uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).

LibPath: here you can configure where are located the openSSL libraries

osIpNone: this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.

osIpDefaultFolder: sets automatically the openSSL path where the libraries should be located for all IDE personalities.

osIpCustomFolder: if this is the option selected, define the full path in the property LibPathCustom.

LibPathCustom: when LibPath = osIpCustomFolder define here the full path where are located the openSSL libraries.

UnixSymLinks: enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):

osIsSymLinksDefault: by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).

osIsSymLinksLoadFirst: Load SymLinks and do before trying to load the version libraries.

osIsSymLinksLoad: Load SymLinks after trying to load the version libraries.

osIsSymLinksDontLoad: don't load the SymLinks.

SChannel_Options: allows you to use a certificate from Windows Certificate Store.

CertHash: is the certificate Hash. You can find the certificate Hash running a dir command in powershell.

CipherList: here you can set which Ciphers will be used (separated by ":"). Example: CALG_AES_256:CALG_AES_128

CertStoreName: the store name where is stored the certificate. Select one of below:

scsnMY (the default)

scsnCA

scsnRoot

scsnTrust

CertStorePath: the store path where is stored the certificate. Select one of below:

scspStoreCurrentUser (the default)

scspStoreLocalMachine

Proxy Options

Use this property to configure the connections through a proxy.

Enabled: set to true to enable proxy connections.
Host: Proxy server address
Port: Proxy server port
UserName/Password: Authentication to connect to proxy, only if required.
ProxyType: the following proxies are supported:

- HTTP
- Socks4
- Socks4A
- Socks5

Log

If Log property is enabled it saves socket messages to a specified log file, useful for debugging.

Log: enable if you want to save the HTTP requests to a text file.
LogFileName: full path to the filename.

Properties

Other properties that can be used to customize the OpenAPI client:

EncodeBodyAsUTF8: if enabled, the request body is encoded as UTF-8 (by default true). A JSON document is UTF-8 per RFC 8259, a body encoded as ANSI mangles every character outside the ASCII range.

JSON Serialization

The classes (DTOs) generated from the specification expose the following properties to customize how they are written to JSON:

JSONIgnoreEmptyStrings: by default it is False, so a field holding an empty string is written as **"field": ""**. An empty string is a value and the server cannot tell it apart from a field that was never set. Set it to True to leave the empty strings out of the JSON, which is what earlier versions did.
JSONIgnoreNullValues: by default it is True, so a null is not written to the JSON. The nulls are controlled by this property alone, it is independent of JSONIgnoreEmptyStrings. Set it to False to write an explicit null, which is what a JSON Merge Patch (RFC 7386) needs to delete a member.

Response

The result of a request is returned in a **TsgcOpenAPIResponse** object.

ResponseStream: stream where the body of the response is written, assign your own stream to save a download directly to it.
OwnsResponseStream: a stream assigned by the caller is no longer destroyed with the response, the caller keeps the ownership. Set it to True to restore the previous behaviour and let the response free the stream.

Cookie Parameters

A parameter declared as **in: cookie** in the specification is supported. All the cookie parameters of a request are sent together in a single **Cookie** header, as RFC 6265 requires.

Asynchronous Requests

By default the OpenAPI client executes requests synchronously. Use the asynchronous API to run a request on a background thread and be notified through events, which keeps the user interface responsive and allows the request to be cancelled.

HTTP_REQUEST_Async: runs the request on a worker thread. The client takes ownership of the request object.
CancelAsync: cancels the request in progress.
SynchronizeEvents: when True the events are marshalled to the main thread. By default it is False and the events fire in the worker thread.

The result is delivered through the following events:

OnResponse: the request completed, the response is passed as a parameter.
OnError: the request failed, the exception is passed as a parameter.
OnCancel: the request was cancelled by CancelAsync.

The client **cannot be freed from inside its own OnResponse, OnError or OnCancel** event handler. Destroying it waits for the worker thread, which is the very thread running the handler, so the application would hang. A clear error is raised instead, free the client once the event handler has returned.

Events

Find below the list of the events that you can handle when using the OpenAPI Client.

OnBeforeRequest

This event is called before the HTTP request is called. Allows to customize the Parameter names, Headers, security... Find below an example how to replace the name of some parameters.

```
void __fastcall OnBeforeRequestEvent(TObject *Sender, const TsgcOpenAPIRequest *aRequest)
{
    for (int i = 0; i < aRequest->Parameters->Count; i++)
    {
        TsgcOpenAPIParameter *oParameter = aRequest->Parameters->Items[i];
        if (oParameter->_Name == "meta-modified-from")
            oParameter->_Name = "eventDateTime-from";
        if (oParameter->_Name == "meta-modified-to")
            oParameter->_Name = "eventDateTime-to";
    }
}
```

OnUpload

This event is called when a file is uploaded, you can use this event to know the progress of the upload.

OnDownload

This event is called when a file is downloaded, you can use this event to know the progress of the download.

OnSSLVerifyPeer

If verify certificate is enabled, in this event you can verify and decide whether to accept the server certificate.

OnSSLGetHandler

This event is raised before SSL handler is created, you can create here your own SSL Handler (needs to be inherited from TIdServerIOHandlerSSLBase or TIdIOHandlerSSLBase) and set the properties needed

OnSSLAfterCreateHandler

If no custom SSL object has been created, it creates by default using OpenSSL handler. You can access to SSL Handler properties and modify if needed

OpenAPI | Server

The **sgcOpenAPI Server** hosts an HTTP service whose contract is described by an OpenAPI 3.0 specification. Routes are built automatically from the spec's **paths** section, a Swagger UI is exposed out-of-the-box and incoming requests can be validated against the JSON-Schemas declared by each operation. Two components are provided to host the server:

- **TsgcHTTPServer_OpenAPI**: standalone HTTP server. Self-contained, embeds an Indy HTTP server, does not require sgcWebSockets. Recommended when you only need to expose an OpenAPI service.
- **TsgcWSAPIServer_OpenAPI**: compatible with sgcWebSockets. Attaches to an existing **TsgcWebSocketHTTPServer** so the same server handles both WebSocket and HTTP/OpenAPI traffic.

Both components share the same engine, options and events.

TsgcHTTPServer_OpenAPI

Standalone server defined in the unit **sgcHTTPServer_OpenAPI**. Configure the listening port, load an OpenAPI spec and activate the server.

TsgcWSAPIServer_OpenAPI

Compatible with sgcWebSockets, defined in the unit **sgcWebSocket_Server_API_OpenAPI**. Assign an existing **TsgcWebSocketHTTPServer** to the **Server** property so a single server instance can dispatch both WebSocket frames and OpenAPI HTTP requests. Internally it delegates to the same engine used by the standalone component, so the API surface is identical.

Built-in Endpoints

The server automatically serves the following endpoints (each can be disabled via **OpenAPIOptions.Endpoint**):

/openapi.json: returns the loaded OpenAPI specification, controlled by **OpenAPIOptions.Endpoint.ServeSpec**.
/docs: embedded Swagger UI to browse and try the API, controlled by **OpenAPIOptions.Endpoint.ServeSwaggerUI**.
OPTIONS *: CORS preflight responses, sent when **OpenAPIOptions.CORS.Enabled** is True.

OpenAPIOptions

Top-level configuration property, of type **TsgcOpenAPIServer_Options**. Groups three sub-objects:

Endpoint: built-in endpoints and spec loading.
BasePath: optional URL prefix prepended to every route declared in the spec.
SpecFile: if assigned, the file is loaded automatically when the server becomes **Active**.
ServeSpec: if True (default) the loaded spec is served at **/openapi.json**.
ServeSwaggerUI: if True (default) the Swagger UI is served at **/docs**.
Validation: request validation flags, all default to False except EnforceRequired.
ValidateRequest: master switch that enables validation for the whole request. On its own it validates every scope below.
ValidateRequestBody: validate the request body against the JSON-Schema declared in the spec.
ValidateQueryParams: validate the query parameters against their declared schema.
ValidatePathParams: validate the path parameters against their declared schema.
ValidateHeaderParams: validate the header parameters against their declared schema.
ValidateCookieParams: validate the cookie parameters against their declared schema.
EnforceRequired: reject the request when a parameter declared as required is missing. It defaults to True and it is applied in every scope which is enabled, because **required** is part of validating a scope and never a scope of its own.
ValidateRequired: superseded by EnforceRequired, kept so that existing code and .dfm files still load. It can only turn **required** on, it can no longer turn it off.
Warnings: read-only list filled when the specification is loaded, with one line per JSON-Schema keyword the spec uses and this validator does not enforce.
CORS: cross-origin support.

Enabled: if True the CORS headers are emitted on every response and **OPTIONS** preflight requests are answered automatically.
AllowOrigins: value of **Access-Control-Allow-Origin** (default *).
AllowHeaders: value of **Access-Control-Allow-Headers** (default **Content-Type, Authorization**).
AllowMethods: value of **Access-Control-Allow-Methods** (default **GET, POST, PUT, DELETE, PATCH, OPTIONS**).

Request Validation

When only the master switch **ValidateRequest** is set, every scope is validated. When one or more of the scope selectors is set, only the selected scopes are.

- **Header and cookie parameters are validated as well**. A specification that declares a header with **required: true** used to pass silently, now the request is rejected with a **400** response when the header is missing or does not match its schema.
- **required** is enforced in every validation scope which is enabled. Previously, enabling a narrower scope such as ValidateQueryParams switched the required check off while still reporting that validation was on. Use **EnforceRequired** to control it.
- A request body whose media type is **not declared in the content** of the operation is rejected with a **415** response, and that includes a request which arrives without a **Content-Type** header. Previously the body fell back to **application/json**.
- A property marked as **readOnly** is rejected when it is sent in a request body. A **required** entry which names a readOnly property applies to the response only, so it is not enforced on the request.

Validation Warnings

The JSON-Schema keywords which the validator does not enforce are **reported when the specification is loaded**, instead of passing silently. Read **OpenAPIOptions.Validation.Warnings** after LoadFromFile or LoadFromString: it holds one line per keyword, with the location where the keyword was first seen. An empty list means every keyword the specification uses is checked, a list which is not empty names exactly what is not checked.
The keywords which are not enforced are:

- **patternProperties**
- **dependentRequired**, **dependentSchemas** and **dependencies**
- **prefixItems** and **additionalItems**
- **unevaluatedProperties** and **unevaluatedItems**
- **contains**, **minContains** and **maxContains**
- **propertyNames**
- **if** / **then** / **else**
- **discriminator**
- **contentEncoding** and **contentMediaType**

The response is not validated either, only the request is.
On Delphi 7 to Delphi 2010 the **pattern** keyword cannot be enforced, the regular expression engine requires Delphi XE or later. It is reported in the same list when the specification is loaded.

Events

Find below the list of the events that you can handle when using the OpenAPI Server.

OnBeforeRequest: fires before the request is dispatched to **OnRequest**. Set the **Accept** parameter to False to reject the request with a 403 response.
OnRequest: main handler. Inspect **aOperationId** and use **aContext** to read the request and write the response. Set **Handled** to True after producing a response.
OnAfterRequest: fires after the user handler returns. Useful for logging or metrics.
OnAuthenticate: fires before dispatch when authentication is required. Set **Authenticated** to False to reject with a 401 response.
OnValidationError: fires when one or more validation rules fail. Inspect **aErrors** and set **Continue** to False to reject the request with a 400 response.
OnException: fires when an unhandled exception propagates from the user handler. Adjust **aResponseCode** if the default 500 should be replaced.

Spec-First vs Code-First

The server supports two authoring approaches:

- Spec-First: the OpenAPI 3.0 JSON/YAML file is the source of truth; the server discovers routes by parsing it.

- Code-First: a Delphi class annotated with RTTI attributes is the source of truth; the OpenAPI spec is generated at startup. Requires Delphi XE7 or later.

OpenAPI | Server | Spec-First

In the **Spec-First** approach the OpenAPI 3.0 specification (JSON or YAML) is the source of truth: it describes every path, operation, parameter and schema. When the server starts it parses the spec and builds the route table automatically; you only need to provide the business logic for each operation.
This workflow is the recommended one when the API contract already exists, when it is shared with consumers, or when it is maintained outside the Delphi code base.

Loading the Spec

Use **LoadFromFile** to load the spec from disk, or **LoadFromString** to load it from an in-memory string (for example, a spec retrieved from a database or generated at runtime).
Once loaded, the route table is built from the spec's **paths** section. Template segments such as **/pets/{petId}** are recognised automatically and the captured values are exposed via **aContext.PathParamAsXxx**.

Handling Operations

The server dispatches every incoming request to the **OnRequest** event, passing the **operationId** declared in the spec. Use a simple **if/else** chain (or a dictionary lookup) to route the operation to the right handler.

Reading Request Data

The **TsgcOpenAPIServerContext** object exposes typed accessors for every part of the request:

Writing Responses

The context object also provides helpers to write the response. The most common helpers are **RespondJSON** and **RespondError**; for full control over the response, set **aContext.Response.Code** and **aContext.Response.ContentText** directly.

Validation

When the validation flags under **OpenAPIOptions.Validation** are enabled, the server validates every request against the JSON-Schemas declared in the spec before invoking **OnRequest**:
Enable **ValidateRequest** to turn validation on. When none of the flags below is set, every category is validated. The request body is checked against the full JSON Schema declared in the spec, including **type**, **enum**, **required**, **oneOf** / **anyOf** / **allOf**, string **formats** and **$ref** references. When validation fails the server responds with **400** and a body that lists the errors.

- **ValidateRequestBody:** the request body is checked against the schema referenced by the operation's **requestBody**.
- **ValidateQueryParams:** query string parameters are checked against their declared schemas.
- **ValidatePathParams:** path parameters are checked against their declared schemas.
- **ValidateHeaderParams:** header parameters are checked against their declared schemas.
- **ValidateCookieParams:** cookie parameters are checked against their declared schemas.
- **EnforceRequired:** the request is rejected when a parameter declared as required is missing. It defaults to True and it is applied in every scope which is enabled. **ValidateRequired** is superseded by it and can only turn the check on.

When validation fails the **OnValidationError** event fires with the list of errors. Set **Continue** to False to reject the request with a 400 response, or leave it True to let your **OnRequest** handler decide.
A request body whose media type is not declared in the **content** of the operation is rejected with a **415** response, and the keywords which the validator does not enforce are reported in **OpenAPIOptions.Validation.Warnings** when the specification is loaded. Read more in the OpenAPI Server topic.

Security

When **OpenAPIOptions.Security.EnforceSecurity** is enabled, the server enforces the **securitySchemes** and the **security** requirements declared in the spec (both the global requirements and the per-operation ones) before invoking **OnRequest**. A request is accepted when it satisfies at least one security requirement. The schemes inside a requirement are combined with AND, the requirements with OR.
The following scheme types are supported: **API Key** (header, query or cookie), **HTTP Basic**, **HTTP Bearer / JWT** and **OAuth2 / OpenID Connect**. When a request does not provide valid credentials the server responds with **401**. When the credentials are valid but the required scope is missing it responds with **403**.
Bearer tokens are verified as JWT (signature, expiration, issuer, audience and scope) when **JWTSecret** is set (an HMAC secret, or a PEM public key for RS/ES). Use **Issuer**, **Audience** and **ValidateExpiration** to tune the checks.
The following events let your application take the final decision for each scheme:

OnValidateAPIKey: validate an API Key credential.
OnValidateBasic: validate a user name and password.
OnValidateBearer: validate a bearer / JWT token.

Mock Responses

Enable **OpenAPIOptions.Mock.Enabled** to let the server answer automatically with example data taken from the spec, which is useful to prototype or test a contract before the business logic is implemented. When a request is not handled by **OnRequest**, the server builds a response from the operation's **examples**, or synthesizes one from the response schema, and returns it with the status code set in **Mock.StatusCode** (200 by default).

Swagger UI

The server can publish the specification at **/openapi.json** and an interactive **Swagger UI** at **/docs**. For environments without internet access, set **OpenAPIOptions.Endpoint.SwaggerUIAssetsPath** to a local folder that contains the Swagger UI assets (swagger-ui.css and swagger-ui-bundle.js) and they will be served locally instead of from a CDN. Use **SwaggerUIBaseURL** to point to a custom CDN base URL.

Demo

A complete spec-first sample is shipped under **Demos/23.OpenAPI/02.OpenAPI_Server_SpecFirst/**.

OpenAPI | Server | Code-First

In the **Code-First** approach the Delphi class is the source of truth: methods, parameters and RTTI attributes describe the API contract, and the OpenAPI 3.0 specification is generated from them at startup. Once generated, the spec is loaded into the server with **LoadFromString** and the route table is built as if it had come from a file.
Code-First requires **Delphi XE7 or later** because it relies on the new RTTI to inspect attributes. The functionality lives in the unit **sgcHTTP_OpenAPI_Server_CodeFirst** and is guarded by the compiler directive **{$IFDEF DXE7}**.

Defining the Service

Annotate a Delphi class with attributes to describe the API. There are two ways of providing the implementation. The method bodies can be left as stubs that exist only so the RTTI scanner can read their signatures, with the actual logic written in the server's **OnRequest** event, or a **TsgcOpenAPICodeFirstDispatcher** can be attached to the server, so that the annotated methods are invoked directly and their bodies hold the implementation.

Available Attributes

The following attributes are recognised by the scanner. Class-level attributes describe the service as a whole; method-level attributes describe a single operation; parameter-level attributes describe where the value comes from.

sgcServiceContract(name, description, version): class-level. Populates the OpenAPI **info** block.
sgcRoute(path): class-level base path, or method-level route relative to the class base path.
sgcHttpGet, sgcHttpPost, sgcHttpPut, sgcHttpDelete, sgcHttpPatch, sgcHttpHead, sgcHttpOptions: method-level. Declare the HTTP verb of the operation.
sgcSummary(text): method-level. Short summary shown in the spec and the Swagger UI.
sgcDescription(text): method-level. Long description for the spec.
sgcTag(name): method-level. Groups operations under the same tag in the Swagger UI.
sgcResponse(code, description): method-level. Declares a response with its HTTP status code and description. Can be repeated to declare multiple responses. The first 2xx response is the success response. See **Responses** below.
sgcResponse(code, description, type): method-level. Same as above, and publishes the class **type** as the schema of the response body, typically for error responses.
sgcResponse(code, description, type, arrayOf): method-level. When **arrayOf** is **True**, the body is published as an array of the class **type**. See **Responses** below.
sgcResponse(code, description, type, contentType): method-level. Publishes the body under the media type **contentType** instead of **application/json**. See **Responses** below.
sgcResponse(code, description, type, arrayOf, contentType): method-level. Combines both: an array of the class **type** under the media type **contentType**.
sgcOperationId(name): method-level. Sets the operationId of the method, which by default is the method name. Use it when two controllers published together have a method with the same name, for example **ListTasks** in **/api/v1** and in **/api/v2**.
sgcExample(value): property-level. Adds an **example** to the schema of the property. See **Examples** below.
sgcExample(code, json): method-level. Adds an example body to the response with that status code. See **Examples** below.
sgcFromPath, sgcFromQuery, sgcFromBody, sgcFromHeader: parameter-level. Declare where the parameter is read from.
sgcRequired: parameter-level. Marks the parameter as required; missing parameters are reported by validation when **EnforceRequired** is enabled, which is the default.

Generating and Loading the Spec

Use **TsgcOpenAPICodeFirstScanner** to generate the OpenAPI 3.0 specification from the annotated class, then load the result into the server.
To publish several controllers in one specification, pass them all to the **GenerateSpec(const aClasses: array of TClass)** overload. The **info** block is read from the first class. When every class has the same **sgcRoute**, that route is published as **servers[0].url**. Otherwise **servers** is omitted and each path key carries the route of its class, for example **/api/v1/tasks** and **/api/v2/tasks**.
If both classes declare a method **ListTasks**, give at least one of them an **sgcOperationId** attribute so that the operationIds stay unique.

Handling Operations

The scanner derives the **operationId** from the method name: the method **ListTasks** becomes the operation **ListTasks**, unless the method has an **sgcOperationId** attribute. Dispatch in **OnRequest** using the same operationId.

Invoking the Annotated Methods

TsgcOpenAPICodeFirstDispatcher invokes the annotated methods directly, so the manual chain of **if aOperationId = ...** comparisons is no longer needed. The class lives in the same unit **sgcHTTP_OpenAPI_Server_CodeFirst** and requires **Delphi XE7 or later**, like the rest of the Code-First functionality. Create a dispatcher, register an instance of the annotated class as a controller, and assign the dispatcher to the **Dispatcher** property of the server.
The dispatcher implements the interface **IsgcOpenAPIDispatcher**, declared in the unit **sgcHTTP_OpenAPI_Server**. The **Dispatcher** property is published by **TsgcWSAPIServer_OpenAPI**, in the unit **sgcWebSocket_Server_API_OpenAPI**, and by **TsgcHTTPServer_OpenAPI**, in the unit **sgcHTTPServer_OpenAPI**.
RegisterController matches the operations by the operationId that the scanner emits, which is the Delphi method name or the value of its **sgcOperationId** attribute. Register the controller and the spec generated from the same class, and the two line up automatically without any extra mapping.

- **RegisterController(aController, aOwnsController):** scans the instance and registers every annotated method that it exposes.
- **DispatchOperation(aOperationId, aContext):** invokes the method registered for the operationId and returns **True** when the operation has been handled. The server calls it internally, so normally there is no need to call it from user code.
- **IsRegistered(aOperationId):** returns **True** when the dispatcher handles the operation.

Precedence. The dispatcher runs first. Every operation it has registered is handled by the controller method, and anything it has not registered falls through to the **OnRequest** event, and then to the mock response or a **501 Not Implemented** status, exactly as before. Existing **OnRequest** code keeps working unchanged. The dispatcher is opt-in and may implement only part of the API, leaving the rest of the operations to the event.
Parameter binding. Parameter values are read from the request according to the attributes already described: **sgcFromPath**, **sgcFromQuery**, **sgcFromHeader** and **sgcFromBody**. A parameter with no location attribute is auto-detected: it is a path parameter when **{name}** appears in the route template of the method, otherwise it is a query parameter. The supported parameter types are **string**, **Integer**, **Int64**, floating point, **Boolean**, other enumerations, and for **sgcFromBody** either a **string**, which receives the raw body, or a class type, which is deserialized from the body. A parameter typed **TsgcOpenAPIServerContext** is injected by the dispatcher. It is not read from the request, and it is deliberately excluded from the generated specification.
Enumeration parameters. An enumeration parameter binds by member name first, and the comparison ignores case, which is what the generated specification advertises because an enumeration is published as **type: "string"**. Both **?priority=prHigh** and **?priority=PRHIGH** bind to the same member. When the value is not a member name it is treated as an ordinal, so **?priority=2** binds as well. A value that resolves outside the range of the enumeration binds the lowest member instead of producing an invalid value, and an unrecognised string does the same.
Returning a result. When the method is a **function**, the result is serialized to JSON and sent with the success status code, which is taken from the **sgcResponse** attribute of the method when that code is a 2xx code, and is 200 otherwise. A **string** result is sent verbatim, so the method returns ready-made JSON. Numeric, **Boolean** and enumeration results are sent as JSON scalars. A class result is serialized and then freed by the dispatcher, so the method must return an instance whose ownership the dispatcher may take. A **TArray<TClass>** result is sent as a JSON array, and the dispatcher frees every element. See **Responses** below. To answer with an error status and a JSON body, raise **EsgcOpenAPIResponseException**, as described in **Error Responses** below.
Full control. A **procedure** that takes an injected **TsgcOpenAPIServerContext** parameter gets full control of the response. The dispatcher does not write anything, so the method decides the status code and the body itself, using **RespondJSON**, **RespondError** or any other method of the context. This is the pattern for operations that answer 201, 204 or 404 depending on the outcome.
GetTask is a function, so the returned **TTaskDTO** is serialized with status 200 and then freed by the dispatcher. **CreateTask** receives the deserialized body and the request context, and writes the response itself, for example a 201 with the new task or a 400 when the payload is rejected.
DTO classes. The dispatcher deserializes the body and serializes the result using the **public** and **published** properties of the class, in the same formats that the specification publishes. See **Data Types** below.
Ownership. By default **RegisterController** takes ownership of the controller and frees it together with the dispatcher. Pass **aOwnsController** as **False** to keep the instance under your own control, for example when the controller is a form or another component with its own lifetime. Registering two methods that produce the same operationId raises an exception, so give one of them an **sgcOperationId** attribute. The dispatcher itself is **not** reference counted, so the code that creates it is the code that must free it. Assigning it to the **Dispatcher** property does not transfer ownership, and clearing that property does not destroy it. The dispatcher must outlive every server component it is assigned to, so the safe teardown order is to clear **Dispatcher** first and then free the dispatcher.

This behaviour is deliberate. The dispatcher implements **IInterface** with **_AddRef** and **_Release** returning **-1**, so the reference held by the server component never keeps the object alive. If it were reference counted, the natural Delphi code of creating the dispatcher, assigning it to the property and freeing it later would double-free the object, because the component would still hold a reference at that point.
Duplicate endpoints. Two methods that resolve to the same HTTP verb and the same path are rejected. The specification generator and **RegisterController** both raise an exception, and the message names the verb, the path and the two methods, so the clash is obvious without reading the generated document.
The same path served with different verbs is legitimate and is not affected. **GET /tasks** together with **POST /tasks**, or **GET**, **PUT** and **DELETE** all on **/tasks/{taskId}**, is the normal case and keeps working. Only the combination of verb and path has to be unique. The path is the class route followed by the method route, so **GET /api/v1/tasks** and **GET /api/v2/tasks**, declared in two controllers, are different endpoints.
Before this check existed, the generator silently emitted the same verb twice inside a single path item, which is an invalid OpenAPI document, and the dispatcher registered both methods while the route table reached only one of them, leaving the other unreachable.
The check spans every controller already registered, not only the one being registered, so two controllers cannot collide with each other either. When the clash is genuine, give one of the two methods a different **sgcRoute** or a different **sgcHttpMethod** attribute.

Data Types

The scanner maps the Delphi types of parameters, properties and results to OpenAPI schemas as follows. The dispatcher reads and writes the JSON in these same formats, and matches incoming member names ignoring case.

- **string, Char: string**.
- **Integer, Int64: integer**, format **int32** or **int64**.
- **Single, Double: number**, format **float** or **double**.
- **Boolean: boolean**.
- **TDateTime: string**, format **date-time**. Written as UTC with milliseconds, for example **2026-10-01T08:30:00.000Z**.
- **TDate: string**, format **date**, for example **2026-10-01**.
- **TTime: string**, format **time**, for example **08:30:00**.
- **TBytes: string**, format **byte**. The value travels as base64.
- **Enumeration:** a named schema under **components/schemas** with an **enum** list, referenced by **$ref**. Values are sent and read by name.
- **Dynamic array (TArray<T>): array**, with the schema of T in **items**.
- **Class:** an **object** schema under **components/schemas** with its **public** and **published** properties, referenced by **$ref**.

The validation attributes **sgcRequired**, **sgcMinLength**, **sgcMaxLength**, **sgcRange** and **sgcPattern** can be applied to the properties of a class and are published in its schema.

Responses

The result type of the method defines the body of the success response.

- **Class:** published as a **$ref** schema under **application/json**. The dispatcher serializes the returned object and then frees it.
- **TArray<TClass>:** published as an **array** of that schema. The dispatcher serializes the array and frees every element.
- **string:** sent as is, so the method returns ready-made JSON.
- **Simple types:** numeric, **Boolean** and enumeration results are sent as JSON values.

Success code. The first **sgcResponse** with a 2xx code is the success response. It carries the schema of the return type, and it is the status code the dispatcher answers with, so **[sgcResponse(201, 'Created')]** makes the method answer 201. When the method has no 2xx **sgcResponse**, the success response is 200 with the description **Successful response**.
Several responses. A method can have several **sgcResponse** attributes, and all of them are published in declaration order. The overload **sgcResponse(aCode, aDescription, aType: TClass)** publishes that class as the body schema of the response, typically for errors, for example **[sgcResponse(400, 'Bad request', TErrorResponse)]**. When a 2xx **sgcResponse** has its own class, that class replaces the return type in the specification. The attribute itself is documentation only. To send that body at runtime, the method raises **EsgcOpenAPIResponseException**, see **Error Responses** below. When the same code is declared twice, only the first attribute is kept.
Array bodies. The overload **sgcResponse(aCode, aDescription, aType: TClass, aArrayOf: Boolean)** publishes the body as an array of that class, **{"type":"array","items":{"$ref":...}}**, when **aArrayOf** is **True**. This is useful when the method returns a **string** with ready-made JSON, so the specification still describes the body.
Media type. The overload **sgcResponse(aCode, aDescription, aType: TClass, aContentType: string)** publishes the body under that media type instead of **application/json**. A method **sgcExample** for the same code is placed

under the same media type. The overload **sgcResponse(aCode, aDescription, aType, aArrayOf, aContentType)** sets both.
At runtime, when the dispatcher answers with a code that declares a content type, either the success code of the return value or the code of a raised **EsgcOpenAPIResponseException**, the **Content-Type** header is that media type with **; charset=utf-8** added, unless it already has a charset. A method that answers through the context keeps the content type it set.

Error Responses

A controller method can answer with an error status and a JSON body in two ways.
Raising EsgcOpenAPIResponseException. Inside a controller method, **raise EsgcOpenAPIResponseException.Create(aCode, aBody)** answers the request with status **aCode** and **aBody** serialized as JSON. The body follows the same rules and **Naming** callbacks as a return value, so nested classes, arrays, dates, **TBytes** and enumerations are written in the usual formats. The exception owns **aBody** and frees it, so the method must not free it. A **nil** body sends **null**. Any other exception still produces the 500 error response of the server, as before. This is how a method sends the error bodies documented with **[sgcResponse(400, 'Bad request', TErrorResponse)]**.
Answering through the context. A function that also takes a **TsgcOpenAPIServerContext** parameter can call **aContext.RespondJSON** or **aContext.RespondError**. That answer is kept: the dispatcher does not overwrite it with the return value, and frees the returned object, or the objects of a returned array, instead. The read-only **Responded** property of **TsgcOpenAPIServerContext** tells whether a response has already been set.
Character encoding. JSON responses, error responses sent through the context and the generated specification are sent as UTF-8 with **Content-Type: application/json; charset=utf-8**, so text outside Latin-1, for example Polish or Chinese characters, arrives intact.

Examples

The **sgcExample** attribute adds example values to the specification, which the Swagger UI shows next to the schemas and responses.
On a property. [sgcExample('2026-10-01T08:30:00Z')] adds **example** to the schema of the property. The value is converted according to the property type:

- **Integer and floating point:** published as a number.
- **Boolean:** published as **true** or **false**.
- **Array, class and record:** the value is JSON text, for example **[sgcExample('["KsefTokenEncryption"]')]**.
- **Other types:** strings, **TBytes**, dates and enumerations are published as a string.

A value that does not fit the type of the property is published as a string.
On a method. [sgcExample(200, '{"id": 1, "name": "Sample"}')] adds an example body to the response with that code, under **application/json**. Only one example per response code is allowed, and the code must also be a declared response. The default 200 response counts as declared. A value that is not valid JSON is published as a string.
The following example combines property examples, a 201 success response, a 400 error response with its own class and a method example.
The responses of the operation are published as follows.

Customizing Names

By default the specification and the JSON use the Delphi names of classes, properties and enumeration values. To publish different names, create a **TsgcOpenAPICodeFirstNaming** and handle its events. Each event receives **aName** with the Delphi name. Change it to rename the item, or leave it empty to keep the Delphi name.

- **OnSchemaName(Sender; aTypeInfo: PTypeInfo; var aName: string):** names the class and enumeration schemas, the key under **components/schemas** and every **$ref**.
- **OnPropertyName(Sender; aTypeInfo: PTypeInfo; const aPropertyName: string; var aName: string):** names the members of an object. **aTypeInfo** is the class that owns the property.
- **OnEnumValueName(Sender; aTypeInfo: PTypeInfo; aOrdinal: Integer; var aName: string):** names the values of an enumeration.

Assign the same instance to the **Naming** property of both **TsgcOpenAPICodeFirstScanner** and **TsgcOpenAPICodeFirstDispatcher**, so that the specification and the wire agree. Neither of them owns it, so free it after both. When **Naming** is **nil**, which is the default, the Delphi names are used. Incoming member names are still matched ignoring case. The dispatcher calls the handlers from several request threads, so they must not change shared state.
The following example publishes **TPublicKeyCertificate** as **PublicKeyCertificate**, the property **PublicKeyId** as **publicKeyId**, and the enumeration value **pkuKsefTokenEncryption** as **KsefTokenEncryption**.

Demo

A complete code-first sample is shipped under **Demos/23.OpenAPI/01.OpenAPI_Server_CodeFirst/**.

OpenAPI | Amazon AWS

The **sgcOpenAPI Amazon AWS Client** (TsgcOpenAPI_Amazon_Client) has its own OpenAPI Client which inherits from TsgcOpenAPI_Client.

This component has a property called **AmazonOptions** that includes all required configurations to connect to Amazon AWS Servers.

AmazonOptions

In AmazonOptions you can define the required **AccessKey** and **SecretKey** (which must be generated previously from your Amazon Account), to authenticate against the Amazon AWS Servers.

An access key grants programmatic access to your resources. This means that you must guard the access key as carefully as the AWS account root user sign-in credentials.

It's a best practice to do the following:

1. **Create an IAM user**, and then **define that user's permissions** as narrowly as possible.
2. **Create the access key** under that IAM user.

Once you've the credentials, set in the following properties:

- AmazonOptions.AccessKey
- AmazonOptions.SecretKey

The AmazonOptions.JSON property allows defining if the responses are in JSON or XML.

IAM roles, users in AWS IAM Identity Center (successor to AWS Single Sign-On), and federated users have temporary security credentials. Temporary security credentials expire after a defined period of time or when the user ends their session. You can set the token for temporary credentials in the property:

- AmazonOptions.SessionToken

Most common uses

- **Configuration**
  - Amazon AWS Credentials
- **APIs**
  - Amazon AWS S3

sgcOpenAPI AWS SDK

Find below a list of the currently available APIs.

- Access Analyzer
- Alexa For Business
- Amazon API Gateway
- Amazon AppConfig
- Amazon Appflow
- Amazon AppIntegrations Service
- Amazon AppStream
- Amazon Athena

- Amazon Augmented AI Runtime
- Amazon Chime
- Amazon Chime SDK Identity
- Amazon Chime SDK Messaging
- Amazon CloudDirectory
- Amazon CloudFront
- Amazon CloudHSM
- Amazon CloudSearch
- Amazon CloudSearch Domain
- Amazon CloudWatch
- Amazon CloudWatch Application Insights
- Amazon CloudWatch Events
- Amazon CloudWatch Logs
- Amazon CodeGuru Profiler
- Amazon CodeGuru Reviewer
- Amazon Cognito Identity
- Amazon Cognito Identity Provider
- Amazon Cognito Sync
- Amazon Comprehend
- Amazon Connect Contact Lens
- Amazon Connect Customer Profiles
- Amazon Connect Participant Service
- Amazon Connect Service
- Amazon Data Lifecycle Manager
- Amazon Detective
- Amazon DevOps Guru
- Amazon DocumentDB with MongoDB compatibility
- Amazon DynamoDB
- Amazon DynamoDB Accelerator (DAX)
- Amazon DynamoDB Streams
- Amazon EC2 Container Registry
- Amazon EC2 Container Service
- Amazon Elastic Inference
- Amazon Elastic Block Store
- Amazon Elastic Compute Cloud
- Amazon Elastic Container Registry Public
- Amazon Elastic File System
- Amazon Elastic Kubernetes Service
- Amazon Elastic Transcoder
- Amazon ElastiCache
- Amazon Elasticsearch Service
- Amazon EMR
- Amazon EMR Containers
- Amazon EventBridge
- Amazon Forecast Query Service
- Amazon Forecast Service
- Amazon Fraud Detector
- Amazon FSx
- Amazon GameLift
- Amazon Glacier
- Amazon GuardDuty
- Amazon HealthLake
- Amazon Honeycode
- Amazon Import/Export Snowball
- Amazon Inspector
- Amazon Interactive Video Service
- Amazon Kinesis
- Amazon Kinesis Analytics
- Amazon Kinesis Firehose
- Amazon Kinesis Video Signaling Channels
- Amazon Kinesis Video Streams
- Amazon Kinesis Video Streams Archived Media
- Amazon Kinesis Video Streams Media
- Amazon Lex Model Building Service
- Amazon Lex Model Building V2

- Amazon Lex Runtime Service
- Amazon Lex Runtime V2
- Amazon Lightsail
- Amazon Location Service
- Amazon Lookout for Equipment
- Amazon Lookout for Metrics
- Amazon Lookout for Vision
- Amazon Machine Learning
- Amazon Macie
- Amazon Macie 2
- Amazon Managed Blockchain
- Amazon Mechanical Turk
- Amazon MemoryDB
- Amazon Mobile Analytics
- Amazon Neptune
- Amazon OpenSearch Service
- Amazon Personalize
- Amazon Personalize Events
- Amazon Personalize Runtime
- Amazon Pinpoint
- Amazon Pinpoint Email Service
- Amazon Pinpoint SMS and Voice Service
- Amazon Polly
- Amazon Prometheus Service
- Amazon QLDB
- Amazon QLDB Session
- Amazon QuickSight
- Amazon Redshift
- Amazon Rekognition
- Amazon Relational Database Service
- Amazon Route 53
- Amazon Route 53 Domains
- Amazon Route 53 Resolver
- Amazon S3 on Outposts
- Amazon Sagemaker Edge Manager
- Amazon SageMaker Feature Store Runtime
- Amazon SageMaker Runtime
- Amazon SageMaker Service
- Amazon Simple Email Service
- Amazon Simple Notification Service
- Amazon Simple Queue Service
- Amazon Simple Storage Service
- Amazon Simple Systems Manager (SSM)
- Amazon Simple Workflow Service
- Amazon SimpleDB
- Amazon Textract
- Amazon Timestream Query
- Amazon Timestream Write
- Amazon Transcribe Service
- Amazon Translate
- Amazon WorkDocs
- Amazon WorkLink
- Amazon WorkMail
- Amazon WorkMail Message Flow
- Amazon WorkSpaces
- AmazonApiGatewayManagementApi
- AmazonApiGatewayV2
- AmazonMQ
- AmazonMWAA
- AmazonNimbleStudio
- AmplifyBackend
- Application Auto Scaling
- Application Migration Service
- Auto Scaling
- AWS Amplify

- AWS App Mesh
- AWS App Runner
- AWS Application Cost Profiler
- AWS Application Discovery Service
- AWS AppSync
- AWS Audit Manager
- AWS Auto Scaling Plans
- AWS Backup
- AWS Batch
- AWS Budgets
- AWS Certificate Manager
- AWS Certificate Manager Private Certificate Authority
- AWS Cloud Map
- AWS Cloud9
- AWS CloudFormation
- AWS CloudHSM V2
- AWS CloudTrail
- AWS CodeBuild
- AWS CodeCommit
- AWS CodeDeploy
- AWS CodePipeline
- AWS CodeStar
- AWS CodeStar connections
- AWS CodeStar Notifications
- AWS Comprehend Medical
- AWS Compute Optimizer
- AWS Config
- AWS Cost and Usage Report Service
- AWS Cost Explorer Service
- AWS Data Exchange
- AWS Data Pipeline
- AWS Database Migration Service
- AWS DataSync
- AWS Device Farm
- AWS Direct Connect
- AWS Directory Service
- AWS EC2 Instance Connect
- AWS Elastic Beanstalk
- AWS Elemental MediaConvert
- AWS Elemental MediaLive
- AWS Elemental MediaPackage
- AWS Elemental MediaPackage VOD
- AWS Elemental MediaStore
- AWS Elemental MediaStore Data Plane
- AWS Fault Injection Simulator
- AWS Global Accelerator
- AWS Glue
- AWS Glue DataBrew
- AWS Greengrass
- AWS Ground Station
- AWS Health APIs and Notifications
- AWS Identity and Access Management
- AWS Import/Export
- AWS IoT
- AWS IoT 1-Click Devices Service
- AWS IoT 1-Click Projects Service
- AWS IoT Analytics
- AWS IoT Core Device Advisor
- AWS IoT Data Plane
- AWS IoT Events
- AWS IoT Events Data
- AWS IoT Fleet Hub
- AWS IoT Greengrass V2
- AWS IoT Jobs Data Plane
- AWS IoT Secure Tunneling

- AWS IoT SiteWise
- AWS IoT Things Graph
- AWS IoT Wireless
- AWS Key Management Service
- AWS Lake Formation
- AWS Lambda
- AWS License Manager
- AWS Marketplace Catalog Service
- AWS Marketplace Commerce Analytics
- AWS Marketplace Entitlement Service
- AWS MediaConnect
- AWS MediaTailor
- AWS Migration Hub
- AWS Migration Hub Config
- AWS Mobile
- AWS Network Firewall
- AWS Network Manager
- AWS OpsWorks
- AWS OpsWorks CM
- AWS Organizations
- AWS Outposts
- AWS Performance Insights
- AWS Price List Service
- AWS Proton
- AWS RDS DataService
- AWS Resource Access Manager
- AWS Resource Groups
- AWS Resource Groups Tagging API
- AWS RoboMaker
- AWS Route53 Recovery Control Config
- AWS Route53 Recovery Readiness
- AWS S3 Control
- AWS Savings Plans
- AWS Secrets Manager
- AWS Security Token Service
- AWS SecurityHub
- AWS Server Migration Service
- AWS Service Catalog
- AWS Service Catalog App Registry
- AWS Shield
- AWS Signer
- AWS Single Sign-On
- AWS Single Sign-On Admin
- AWS Snow Device Management
- AWS SSO Identity Store
- AWS SSO OIDC
- AWS Step Functions
- AWS Storage Gateway
- AWS Support
- AWS Systems Manager Incident Manager
- AWS Systems Manager Incident Manager Contacts
- AWS Transfer Family
- AWS WAF
- AWS WAF Regional
- AWS WAFV2
- AWS Well-Architected Tool
- AWS X-Ray
- AWSKendraFrontendService
- AWSMarketplace Metering
- AWSServerlessApplicationRepository
- Braket
- CodeArtifact
- EC2 Image Builder
- Elastic Load Balancing
- FinSpace Public API

- FinSpace User Environment Management service
- Firewall Management Service
- Managed Streaming for Kafka
- Managed Streaming for Kafka Connect
- Redshift Data API Service
- Route53 Recovery Cluster
- Schemas
- Service Quotas
- Synthetics

OpenAPI Amazon AWS | Credentials

AWS requires different types of security credentials depending on how you access AWS. For example, you need a user name and password to sign in to the AWS Management Console and you need **access keys** to make **programmatic calls to AWS**.

Considerations

- Be sure to save the following in a secure location: the email address associated with your AWS account, the AWS account ID, the root user password, and your account access keys. If you forget or lose your root user password, you must have access to the email address associated with your account in order to reset it. If you forget or lose your access keys, you must sign into your account to create new ones.
- We strongly recommend that you create an IAM user with administrator permissions to use for everyday AWS tasks and lock away the password and access keys for the root user. Use the root user only for the tasks that are restricted to the root user.
- Security credentials are account-specific. If you have access to multiple AWS accounts, you have separate credentials for each account.
- Do not provide your AWS credentials to a third party.

Programmatic access

You must provide your AWS access keys to make programmatic calls to AWS.

When you create your access keys, you create the access key ID (for example, AKIAIOSFODNN7EXAMPLE) and secret access key (for example, wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY) as a set. The secret access key is available for download only when you create it. If you don't download your secret access key or if you lose it, you must create a new one.

You can assign up to two access keys per user (root user or IAM user). Having two access keys is useful when you want to rotate them. When you disable an access key, you can't use it, but it counts toward your limit of two access keys. After you delete an access key, it's gone forever and can't be restored, but it can be replaced with a new access key.

To manage access keys when signed in as the root user

1. Sign in to the AWS Management Console as the root user.
2. In the navigation bar on the upper right, choose your account name or number and then choose **My Security Credentials**.
3. Expand the **Access keys (access key ID and secret access key)** section.
4. Do one of the following:
    - To create an access key, choose **Create New Access Key**. If you already have two access keys, this button is disabled and you must delete an access key before you can create a new one. When prompted, choose either **Show Access Key** or **Download Key File**. This is your only opportunity to save your secret access key. After you've saved your secret access key in a secure location, choose **Close**.
    - To deactivate an access key, choose **Make Inactive**. When prompted for confirmation, choose **Deactivate**. A deactivated access key still counts toward your limit of two access keys.
    - To activate an access key, choose **Make Active**.
    - To delete an access key when you no longer need it, copy the access key ID and then choose **Delete**. Before you can delete the access key, you must choose **Deactivate**. We recommend that you verify that the access key is no longer in use before you permanently delete it. To confirm deletion, paste the access key ID in the text input field and then choose **Delete**.

To manage access keys when signed in as an IAM user

1. Sign in to the AWS Management Console as an IAM user.
2. In the navigation bar on the upper right, choose your user name and then choose **My Security Credentials**.
3. Do one of the following:

- To create an access key, choose **Create access key**. If you already have two access keys, this button is disabled and you must delete an access key before you can create a new one. When prompted, choose either **Show secret access key** or **Download .csv file**. This is your only opportunity to save your secret access key. After you've saved your secret access key in a secure location, choose **Close**.
- To deactivate an access key, choose **Make inactive**. When prompted for confirmation, choose **Deactivate**. A deactivated access key still counts toward your limit of two access keys.
- To activate an access key, choose **Make active**. When prompted for confirmation, choose **Make active**.
- To delete an access key when you no longer need it, copy the access key ID and then choose **Delete**. This deactivates the access key. We recommend that you verify that the access key is no longer in use before you permanently delete it. To confirm deletion, paste the access key ID in the text input field and then choose **Delete**.

sgcOpenAPI Configuration

Once you have your own AWS Access Keys, you must configure in the OpenAPI Amazon Client before you do any Request to the Amazon AWS Servers.

```
GetOpenAPIClient->AmazonOptions->AccessKey = "AKIAIOSFODNN7EXAMPLE";
GetOpenAPIClient->AmazonOptions->SecretKey = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY";
```

OpenAPI Amazon AWS | S3

Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. Customers of all sizes and industries can use Amazon S3 to store and protect any amount of data for a range of use cases, such as data lakes, websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics. Amazon S3 provides management features so that you can optimize, organize, and configure access to your data to meet your specific business, organizational, and compliance requirements.

ListBuckets

```
GetOpenAPIClient->AmazonOptions->AccessKey = "AKIAIOSFODNN7EXAMPLE";
GetOpenAPIClient->AmazonOptions->SecretKey = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY";
GetOpenAPIClient->ListBuckets();
```

GetObject

```
GetOpenAPIClient->AmazonOptions->AccessKey = "AKIAIOSFODNN7EXAMPLE";
GetOpenAPIClient->AmazonOptions->SecretKey = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY";
GetOpenAPIClient->GetObject("bucket_name");
```

PutObject

```
GetOpenAPIClient->AmazonOptions->AccessKey = "AKIAIOSFODNN7EXAMPLE";
GetOpenAPIClient->AmazonOptions->SecretKey = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY";
GetOpenAPIClient->PutObject("bucket_name", "MyFile.txt", "payload");
```

OpenAPI | Google Cloud

The **sgcOpenAPI Google Client** (TsgcOpenAPI_Google_Client) has its own OpenAPI Client which inherits from TsgcOpenAPI_Client.

This component has a property called **GoogleOptions** that includes all required configurations to connect to Google Cloud Servers.

GoogleOptions

The OpenAPI Google client allows you to authenticate using the following methods:

1. **OAuth2 Code:** is interactive, which means requires the intervention of the user.
2. **JWT (service accounts):** is non-interactive, so can run as a service for example.

The authentication is configured in the property GoogleOptions.Authentication, allows the following values:

- **oagaOAuth2:** interactive.
- **oagaJWT:** non-interactive. You can import the settings from a JSON file, using the method **LoadSettingsFromFile**. This method will fill the following properties automatically:
  - **ClientEmail**
  - **PrivateKeyId**
  - **PrivateKey**
  - **ServiceAccountOptions:** when running the client using a service account, the following properties are required:
    - **TokenURI:** by default the value is "https://oauth2.googleapis.com/token", but the value is updated when using the method LoadSettingsFromFile.
    - **Subject:** this is the email account when using Domain-Wide delegation
    - **Scopes:** a list of the scopes.

Most common uses

- **Configuration**
  - Google Cloud OAuth2
  - Google Cloud Service Accounts
- **APIs**
  - Google Cloud PubSub
  - Google Cloud Calendar
  - Google Cloud Drive

sgcOpenAPI Google Cloud SDK

Find below a list of the currently available APIs.

- Abusive Experience Report API
- Accelerated Mobile Pages (AMP) URL API
- Access Approval API
- Access Context Manager API
- Ad Exchange Buyer API
- Ad Exchange Buyer API II
- Ad Experience Report API
- Admin SDK API
- AdMob API
- AdSense Host API
- AdSense Management API
- AI Platform Training & Prediction API

- Analytics Reporting API
- Android Device Provisioning Partner API
- Android Management API
- API Discovery Service
- API Gateway API
- API Keys API
- Apigee API
- App Engine Admin API
- Apps Script API
- Area120 Tables API
- Artifact Registry API
- Assured Workloads API
- Authorized Buyers Marketplace API
- Backup for GKE API
- Bare Metal Solution API
- BigQuery API
- BigQuery Connection API
- BigQuery Data Transfer API
- BigQuery Reservation API
- Binary Authorization API
- Blogger API v3
- Books API
- Calendar API
- Campaign Manager 360 API
- Certificate Authority API
- Certificate Manager API
- Chrome Management API
- Chrome Policy API
- Chrome UX Report API
- Chrome Verified Access API
- Cloud Asset API
- Cloud AutoML API
- Cloud Bigtable Admin API
- Cloud Billing API
- Cloud Billing Budget API
- Cloud Build API
- Cloud Channel API
- Cloud Composer API
- Cloud Data Fusion API
- Cloud Data Loss Prevention (DLP) API
- Cloud Dataplex API
- Cloud Dataproc API
- Cloud Datastore API
- Cloud Debugger API
- Cloud Deployment Manager V2 API
- Cloud DNS API
- Cloud Document AI API
- Cloud Domains API
- Cloud Filestore API
- Cloud Firestore API
- Cloud Functions API
- Cloud Healthcare API
- Cloud Identity API
- Cloud Identity-Aware Proxy API
- Cloud IDS API
- Cloud IoT API
- Cloud Key Management Service (KMS) API
- Cloud Life Sciences API
- Cloud Logging API
- Cloud Memorystore for Memcached API
- Cloud Monitoring API
- Cloud Natural Language API
- Cloud OS Login API
- Cloud Private Catalog
- Cloud Private Catalog Producer

- Cloud Pub/Sub API
- Cloud Resource Manager API
- Cloud Run Admin API
- Cloud Runtime Configuration API
- Cloud Scheduler API
- Cloud Search API
- Cloud Shell API
- Cloud Source Repositories API
- Cloud Spanner API
- Cloud Speech-to-Text API
- Cloud SQL Admin API
- Cloud Storage for Firebase API
- Cloud Storage JSON API
- Cloud Talent Solution API
- Cloud Tasks API
- Cloud Testing API
- Cloud Text-to-Speech API
- Cloud Tool Results API
- Cloud TPU API
- Cloud Trace API
- Cloud Translation API
- Cloud Video Intelligence API
- Cloud Vision API
- Compute Engine API
- Connectors API
- Contact Center AI Insights API
- Container Analysis API
- Content API for Shopping
- Custom Search API
- Data Labeling API
- Data pipelines API
- Database Migration API
- Dataflow API
- Dataproc Metastore API
- Datastream API
- Dialogflow API
- Digital Asset Links API
- Display & Video 360 API
- Domains RDAP API
- DoubleClick Bid Manager API
- Drive Activity API
- Drive API
- Enterprise License Manager API
- Error Reporting API
- Essential Contacts API
- Eventarc API
- Fact Check Tools API
- Firebase App Check API
- Firebase Cloud Messaging API
- Firebase Cloud Messaging Data API
- Firebase Dynamic Links API
- Firebase Hosting API
- Firebase Management API
- Firebase ML API
- Firebase Realtime Database Management API
- Firebase Rules API
- Fitness API
- Game Services API
- Genomics API
- GKE Hub API
- Gmail API
- Gmail Postmaster Tools API
- Google Analytics Admin API
- Google Analytics API
- Google Analytics Data API

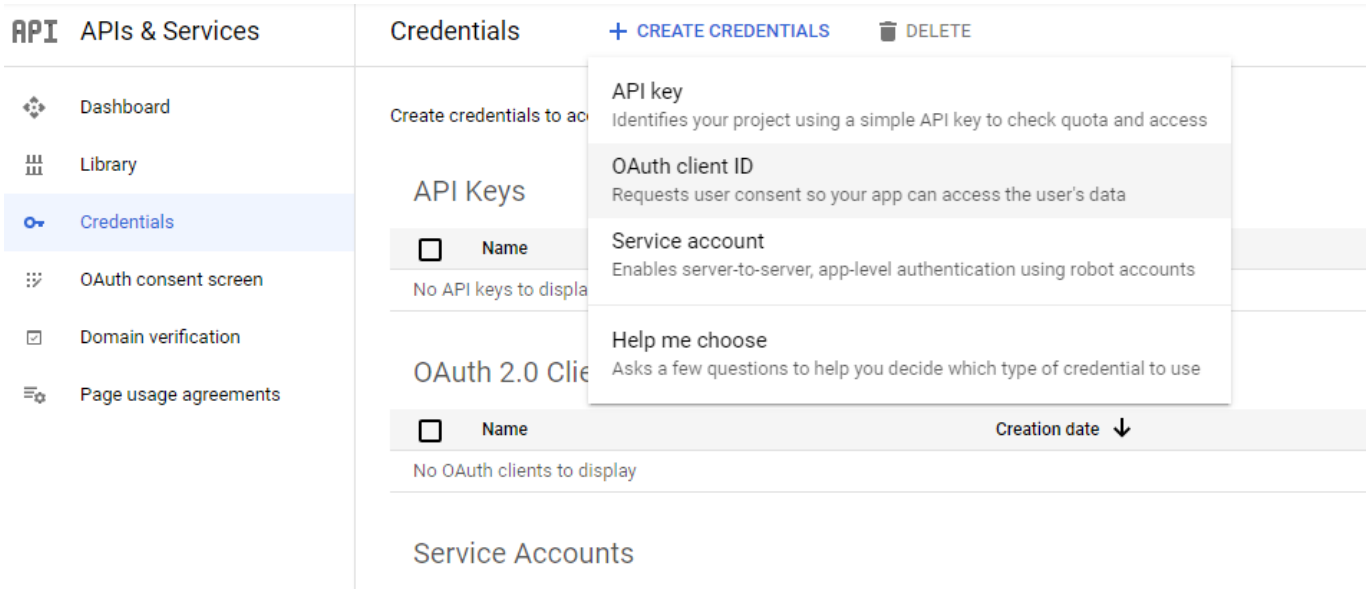
- Google Chat API
- Google Civic Information API
- Google Classroom API
- Google Cloud Data Catalog API
- Google Cloud Deploy API
- Google Cloud Memorystore for Redis API
- Google Cloud Support API
- Google Docs API
- Google Forms API
- Google Identity Toolkit API
- Google Keep API
- Google Mirror
- Google My Business API
- Google OAuth2 API
- Google Pay Passes API
- Google Play Android Developer API
- Google Play Custom App Publishing API
- Google Play Developer Reporting API
- Google Play EMM API
- Google Play Game Management
- Google Play Game Services
- Google Play Game Services Publishing API
- Google Play Integrity API
- Google Search Console API
- Google Sheets API
- Google Site Verification API
- Google Slides API
- Google Vault API
- Google Workspace Alert Center API
- Google Workspace Reseller API
- Google+ API
- Groups Migration API
- Groups Settings API
- HomeGraph API
- IAM Service Account Credentials API
- Idea Hub API
- Identity and Access Management (IAM) API
- Indexing API
- Knowledge Graph Search API
- Kubernetes Engine API
- Library Agent API
- Local Services API
- Managed Service for Microsoft Active Directory API
- Manufacturer Center API
- My Business Account Management API
- My Business Business Calls API
- My Business Business Information API
- My Business Lodging API
- My Business Notifications API
- My Business Place Actions API
- My Business Q&A API
- My Business Verifications API
- Network Connectivity API
- Network Management API
- Network Security API
- Network Services API
- Notebooks API
- On-Demand Scanning API
- Organization Policy API
- OS Config API
- PageSpeed Insights API
- Payments Reseller Subscription API
- People API
- Perspective Comment Analyzer API
- Playable Locations API

- Policy Analyzer API
- Policy Simulator API
- Policy Troubleshooter API
- Poly API
- Proximity Beacon API
- Pub/Sub Lite API
- Real-time Bidding API
- reCAPTCHA Enterprise API
- Recommendations AI (Beta)
- Recommender API
- Remote Build Execution API
- Replica Pool
- Resource Settings API
- Retail API
- Safe Browsing API
- SAS Portal API
- SAS Portal API (Testing)
- Search Ads 360 API
- Search Console API
- Secret Manager API
- Security Command Center API
- Security Token Service API
- Semantic Tile API
- Service Broker
- Service Consumer Management API
- Service Control API
- Service Directory API
- Service Management API
- Service Networking API
- Service Usage API
- Smart Device Management API
- Stackdriver Profiler API
- Storage Transfer API
- Street View Publish API
- Tag Manager API
- Tasks API
- Traffic Director API
- Transcoder API
- Version History API
- VM Migration API
- Web Fonts Developer API
- Web Risk API
- Web Security Scanner API
- Workflow Executions API
- Workflows API
- YouTube Analytics API
- YouTube Data API v3
- YouTube Reporting API

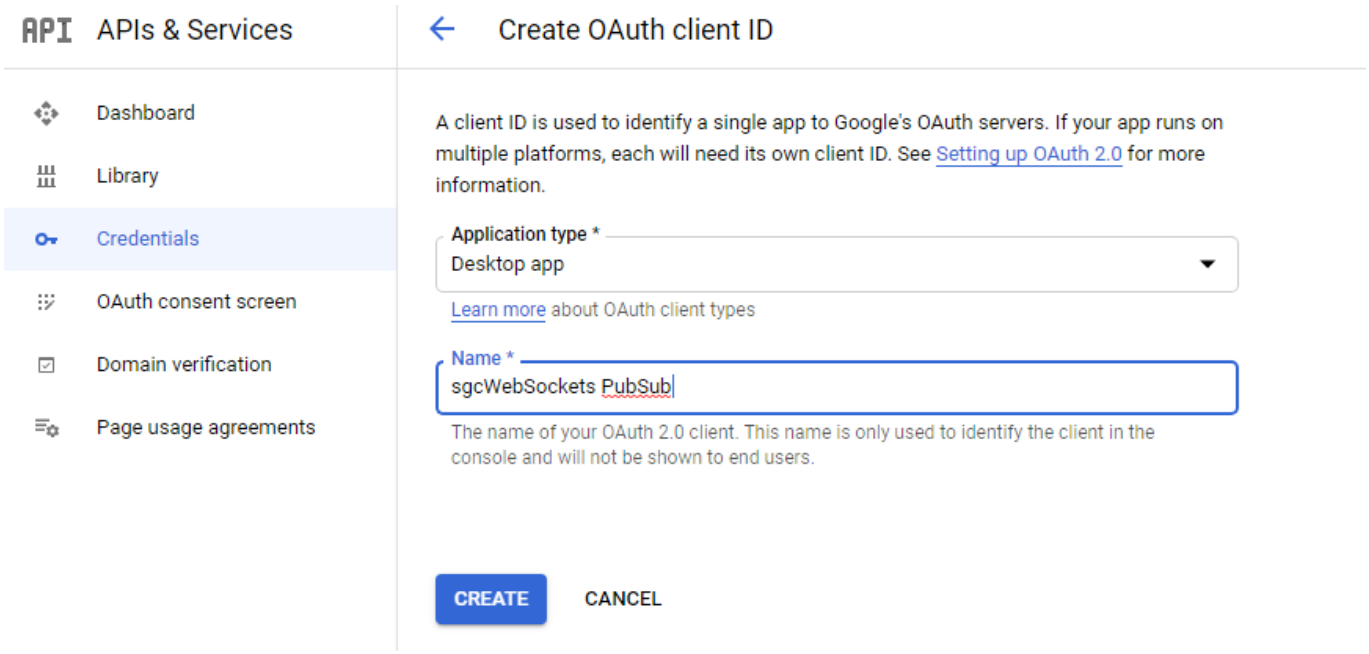
OpenAPI Google Cloud | OAuth2

In order to use the OpenAPI Google Cloud components and Authenticate using OAuth2, first you must obtain the OAuth2 Key from Google Cloud.
Find below the steps to get Google OAuth2 Keys and how to configure in our PubSub sample application.

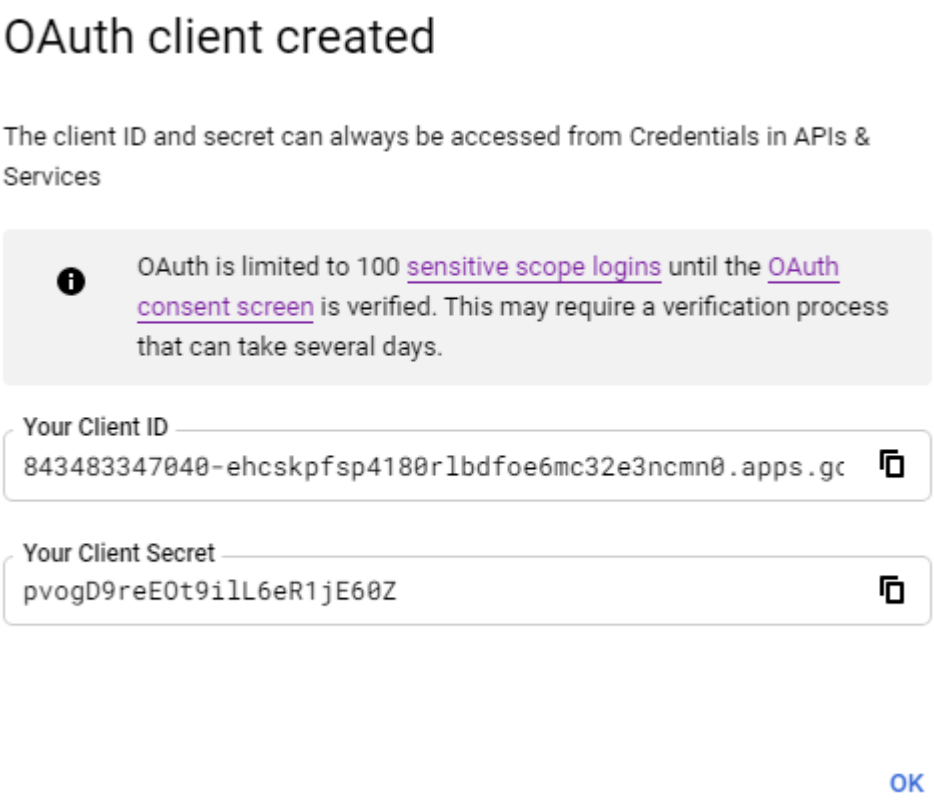
First **login** to your **Google Cloud Account** and use an existing project or create a new one.
After that, go to **Credentials** menu and press the button **CREATE CREDENTIALS**, select the option **OAuth Client ID**.



Select your application type and set a description name

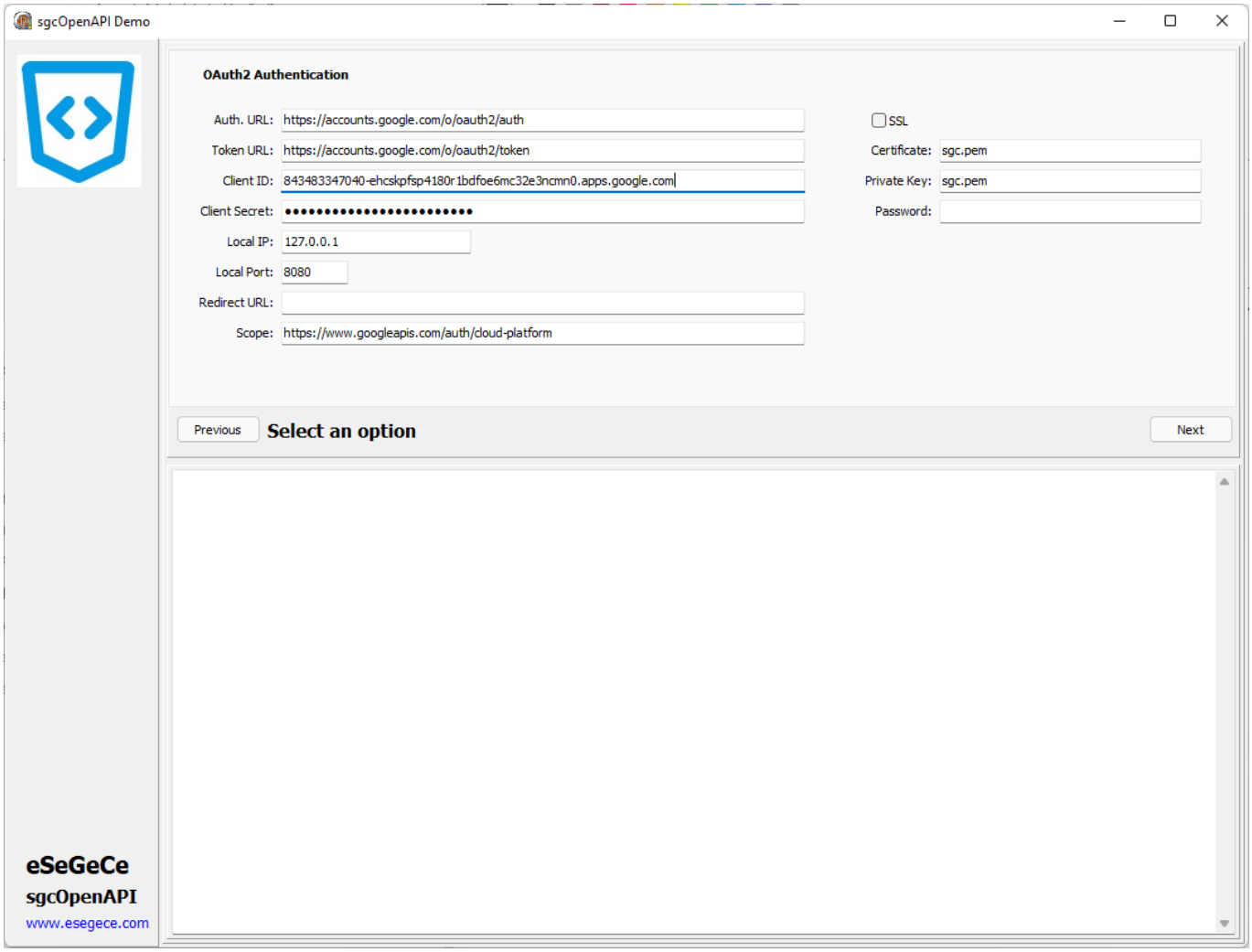


If successful, you will get your **Client Id** and **Client Secret**.



Don't share your OAuth2 data with anyone!

Now copy to the OpenAPI Google Cloud sample



Read more about OAuth2 Configuration.

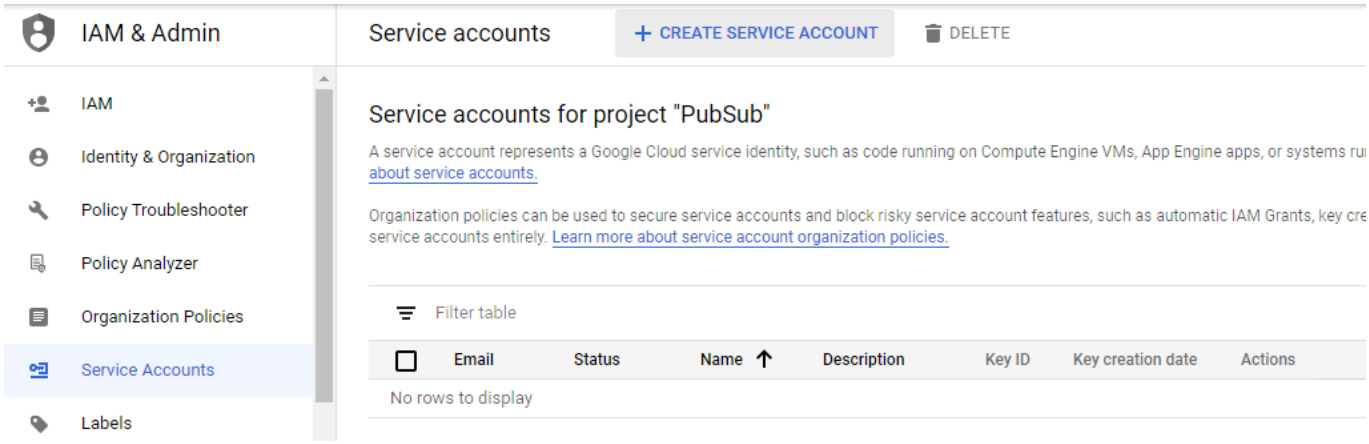
Once you are authenticated, you can **re-authenticate** calling first the method **ClearOAuth2Token** (clear all internal OAuth2 Tokens) and then call any OpenAPI requests, a new web-browser will be shown to re-authenticate against google servers.

OpenAPI Google Cloud | Service Accounts

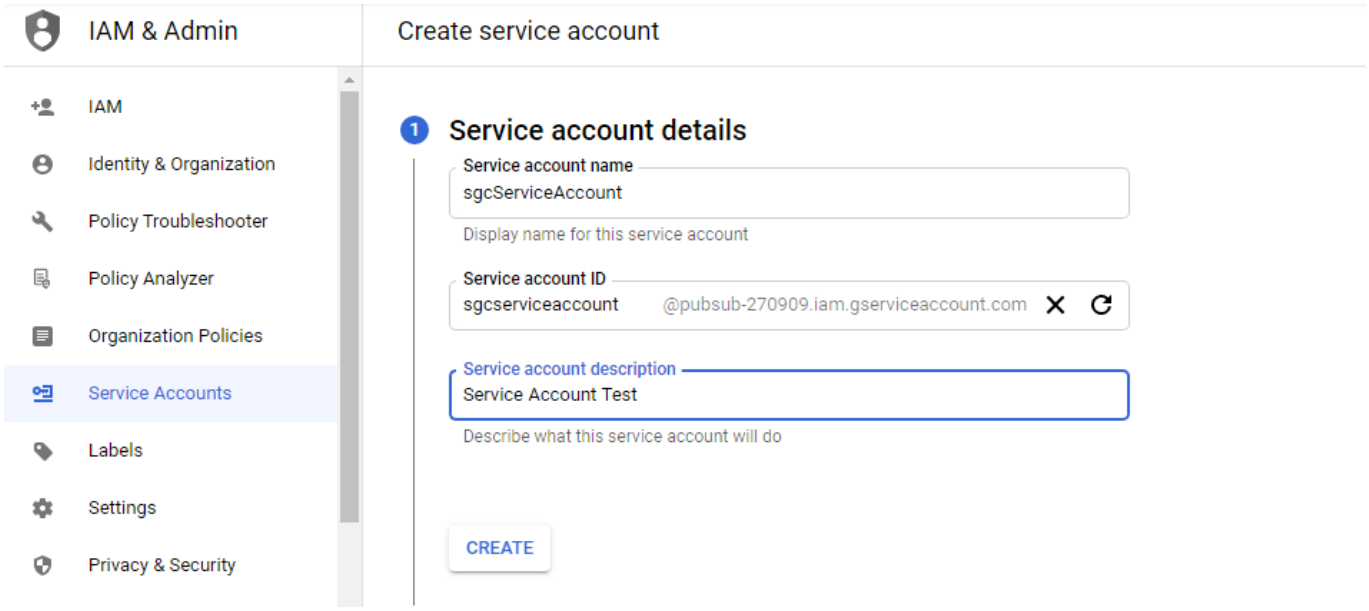
In order to use the **OpenAPI Google Cloud components** and **Authenticate using Service Accounts**, first you must obtain the Private Key Certificate from Google Cloud.

Find below the steps to get Google Private Key Certificate and how to configure in our sample application.

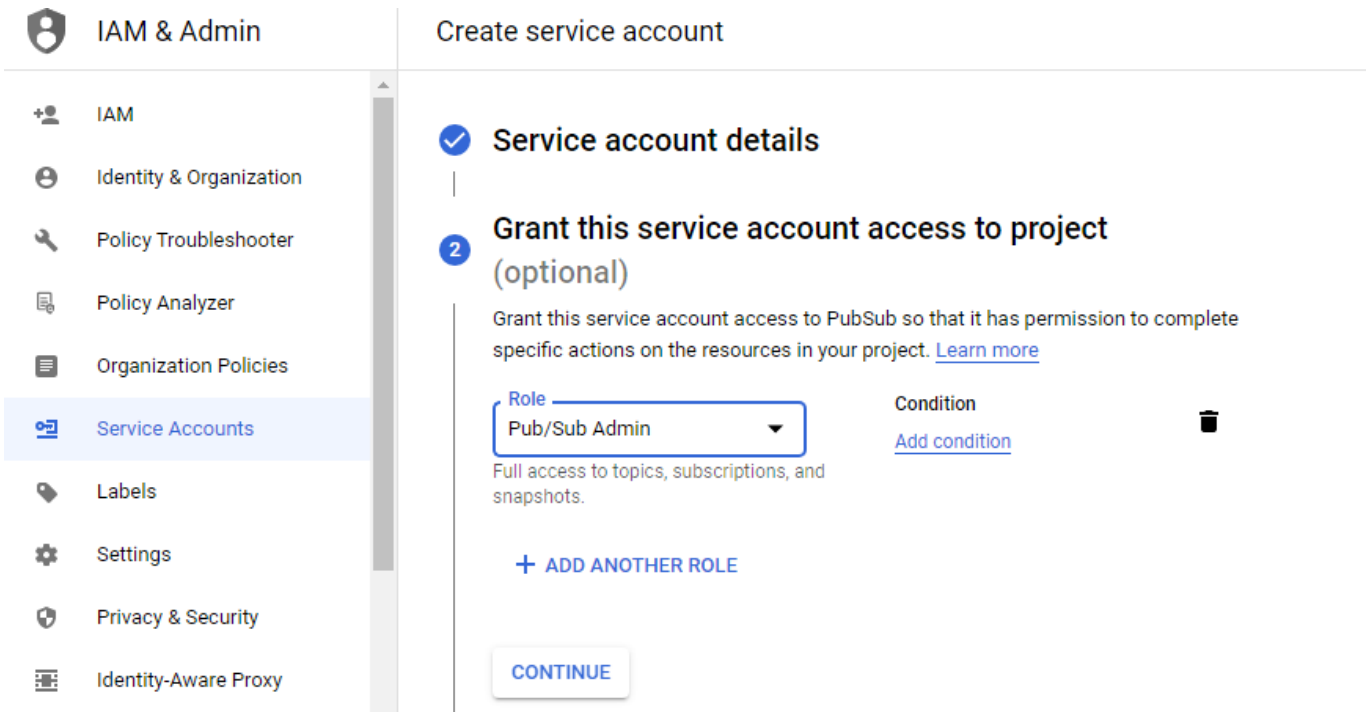
First **login** to your **Google Cloud Account** and use an existing project or create a new one.



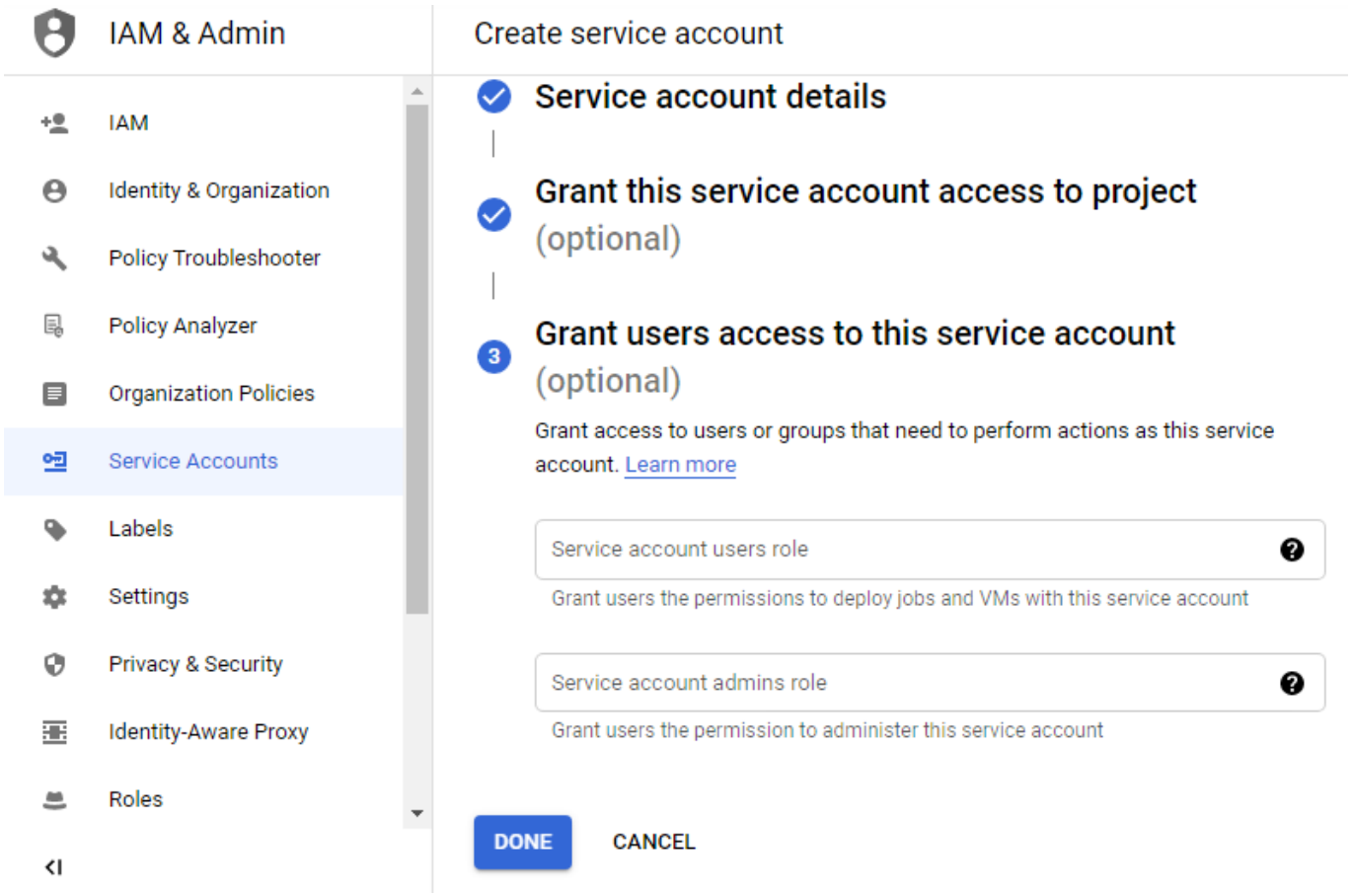
Select **CREATE SERVICE ACCOUNT** and a new page will be shown where you must set the service account name and description



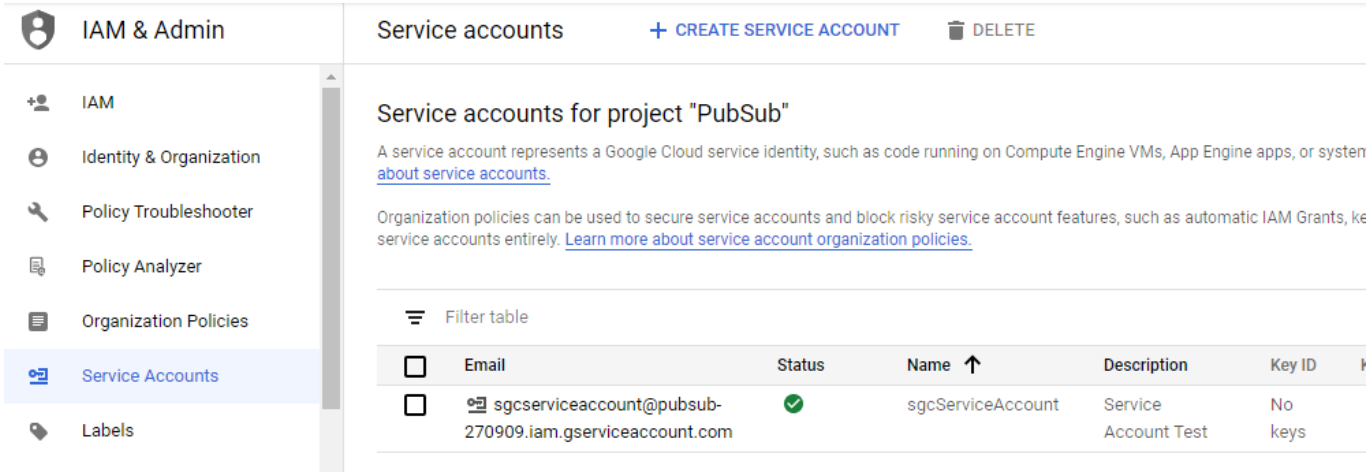
Then select at least one Role, I select PubSub Admin to allow the client publish and subscribe topics, but you can select other role with less privileges



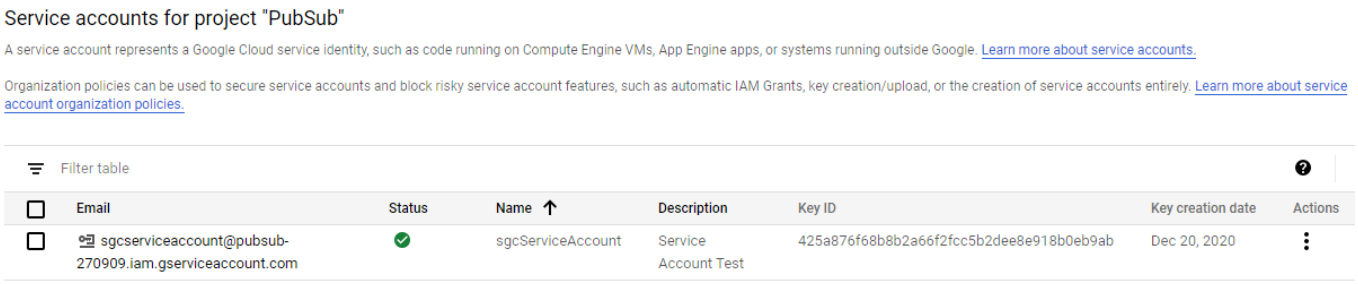
Press CONTINUE and finally you can grant access to other users



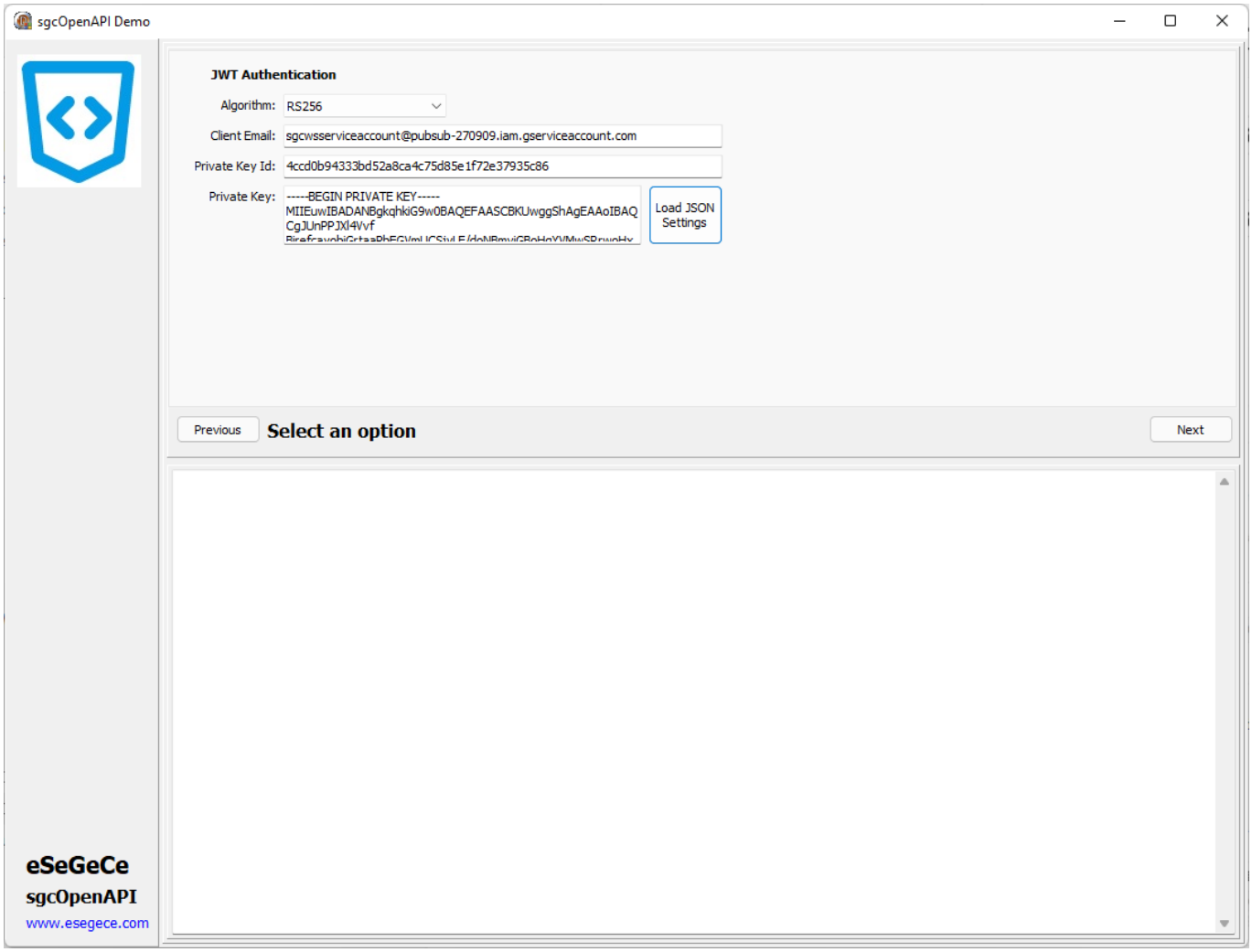
Press DONE when you finish and a new record will be shown



The next step is to create a new Key, so select the option Create Key in actions column. Select JSON to download the configuration in JSON format and a new Key will be created



Finally you only need to fill the data provided by google in the OpenAPI PubSub client. You can use **LoadSettings-FromFile** to load the configuration JSON file.



Domain-Wide Delegation

If you have a Google Workspace account, an administrator of the organization can authorize an application to access user data on behalf of users in the Google Workspace domain. For example, an application that uses the **Google Calendar API** to add events to the calendars of all users in a Google Workspace domain would use a service account to access the Google Calendar API on behalf of users. Authorizing a service account to access data on behalf of users in a domain is sometimes referred to as "delegating domain-wide authority" to a service account.

To delegate domain-wide authority to a service account, a super administrator of the Google Workspace domain must complete the following steps:

- From your Google Workspace domain's Admin console, go to **Main menu menu > Security > Access** and **data control > API Controls**.
- In the **Domain wide delegation pane**, select **Manage Domain Wide Delegation**.
- Click **Add new**.
- In the **Client ID** field, enter the service account's **Client ID**. You can find your service account's client ID in the Service accounts page.
- In the **OAuth scopes (comma-delimited)** field, enter the list of scopes that your application should be granted access to. For example, if your application needs domain-wide full access to the Google Drive API and the Google Calendar API, enter: https://www.googleapis.com/auth/drive, https://www.googleapis.com/auth/calendar.
- Click **Authorize**.

Once you've linked and authorized the workspace account, configure the property GoogleOptions.ServiceAccountOptions from the OpenAPI client:

- **Subject:** is the workspace email account linked to the service account. Example: youremail@domain.com
- **Scopes:** list of scopes. Example: https://www.googleapis.com/auth/calendar

- **TokenURI:** by default is https://oauth2.googleapis.com/token.

OpenAPI Google Cloud | PubSub

Pub/Sub allows services to communicate asynchronously, with latencies on the order of 100 milliseconds.

Pub/Sub is used for streaming analytics and data integration pipelines to ingest and distribute data. It is equally effective as a messaging- oriented middleware for service integration or as a queue to parallelize tasks.

List Projects by Topic (OAuth2)

```
GetOpenAPIClient->GoogleOptions->Authentication = oagaOAuth2;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->AuthURL = "https://accounts.google.com/o/oa
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->TokenURL = "https://accounts.google.com/o/o
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientId = "google client id";
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientSecret = "google client secret";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->IP = "127.0.0.1";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->Port = 8080;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->Scope->Text = "https://www.googleapis.com/a
GetOpenAPIClient->ListV1TopicsByProject("projects/pubsub-270909");
```

List Projects by Topic (Service Accounts)

```
GetOpenAPIClient->GoogleOptions->Authentication = oagaJWT;
GetOpenAPIClient->LoadSettingsFromFile("google.json");
GetOpenAPIClient->ListV1TopicsByProject("projects/pubsub-270909");
```

OpenAPI Google Cloud Calendar

The Calendar API lets you integrate your app with Google Calendar, creating new ways for you to engage your users.

List Events By Calendar (OAuth2)

```
GetOpenAPIClient->GoogleOptions->Authentication = oagaOAuth2;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->AuthURL = "https://accounts.google.com/o/oa
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->TokenURL = "https://accounts.google.com/o/c
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientId = "google client id";
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientSecret = "google client secret";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->IP = "127.0.0.1";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->Port = 8080;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->Scope->Text = "https://www.googleapis.com/a
GetOpenAPIClient->ListCalendarsEventsByCalendarId("email@mydomain.com", true, "json", 0, "");
```

List Events By Calendar (Service Account)

```
GetOpenAPIClient->GoogleOptions->Authentication = oagaJWT;
GetOpenAPIClient->LoadSettingsFromFile("google.json");
GetOpenAPIClient->Authentication->ServiceAccountOptions->Subject = "email@mydomain.com";
GetOpenAPIClient->Authentication->ServiceAccountOptions->Scopes->Text := "https://www.googleapis.com/auth/calenda
GetOpenAPIClient->ListCalendarsEventsByCalendarId("email@mydomain.com", true, "json", 0, "");
```

Insert Calendar Event

```
GetOpenAPIClient->InsertCalendarsEventsByCalendarId("email@mydomain.com", "{\"summary\":\"Test Event\",\"descript
```

OpenAPI Google Cloud | Drive

Use the unit sgcOpenAPI_googleapis.com_drive_helpers if you want to upload and/or download files from your Google Drive Account.

Upload Files

Use the method UploadFile to upload a new file to your google drive account. The Event **OnUpload** allows you to know the progress of the upload.

```
void __fastcall UploadFileToGoogleDrive()
{
    TOpenDialog *oDialog = new TOpenDialog(NULL);
    try
    {
        if (oDialog->Execute())
        {
            GetOpenAPIClient()->OnUpload = OnUploadEvent;
            ShowMessage(GetOpenAPIClient()->UploadFile(oDialog->FileName));
        }
    }
    __finally
    {
        delete oDialog;
    }
}
void __fastcall OnUploadEvent(TObject *Sender, TsgcOpenAPIProgressState aState, __int64 aValue)
{
    switch (aState)
    {
        case oahpStart:
            ProgressBar1->Visible = true;
            ProgressBar1->Position = 0;
            ProgressBar1->Max = aValue;
            break;
        case oahpProgress:
            ProgressBar1->Visible = true;
            ProgressBar1->Position = aValue;
            break;
        case oahpEnd:
            ProgressBar1->Visible = false;
            ProgressBar1->Position = 0;
            ProgressBar1->Max = 0;
            break;
    }
    Application->ProcessMessages();
}
```

Download Files

Use the method **DownloadFile** to download files from your Google Drive Account. The Event **OnDownload** allows you to know the progress of the download.

```
void __fastcall DownloadFileFromGoogleDrive()
{
    TSaveDialog *oSaveDialog = new TSaveDialog(NULL);
    try
    {
        if (oSaveDialog->Execute())
        {
            GetOpenAPIClient()->OnDownload = OnDownloadEvent;
            GetOpenAPIClient()->DownloadFile(txtDownloadFileId->Text, oSaveDialog->FileName);
        }
    }
    __finally
    {
        delete oSaveDialog;
    }
```

```
}
void __fastcall OnDownloadEvent(TObject *Sender, TsgcOpenAPIProgressState aState, __int64 aValue)
{
    switch (aState)
    {
        case oahpStart:
            ProgressBar1->Visible = true;
            ProgressBar1->Position = 0;
            ProgressBar1->Max = aValue;
            break;
        case oahpProgress:
            ProgressBar1->Visible = true;
            ProgressBar1->Position = aValue;
            break;
        case oahpEnd:
            ProgressBar1->Visible = false;
            ProgressBar1->Position = 0;
            ProgressBar1->Max = 0;
            break;
    }
    Application->ProcessMessages();
}
```

List Folders

List all the folders in the drive account. Use the **ListFiles** method and pass the parameter q as "application/vnd.google-apps.folder".

```
GetOpenAPIClient().ListFiles("", true, "", false, false, 100, "", false, false, "", "", "", "", "", "", "", "", "
```

OpenAPI | Microsoft

The **sgcOpenAPI Microsoft Client** (TsgcOpenAPI_Microsoft_Client) has its own OpenAPI Client which inherits from TsgcOpenAPI_Client.

This component has a property called **MicrosoftOptions** that includes all required configurations to connect to Microsoft Servers.

MicrosoftOptions

The OpenAPI Microsoft client allows you to authenticate using the following methods:

1. **OAuth2 Code:** is interactive, which means requires the intervention of the user.
2. **OAuth2 Credentials:** is non-interactive, so can run as a service for example.

The authentication is configured in the property MicrosoftOptions.Authentication, allows the following values:

- **oamaOAuth2Code:** interactive.
- **oamaOAuth2Credentials:** non-interactive.

Other properties required by Microsoft are the following:

- **TenantId**: it's a value that identifies your account, you can find in your Microsoft/Azure account.

Most common uses

- **Configuration**
  - Microsoft Get Tenant
  - Microsoft Register Application
  - Microsoft OAuth2 Code
  - Microsoft OAuth2 Credentials
- **APIs**
  - Microsoft Graph

sgcOpenAPI Microsoft APIs

Find below a list of the currently available APIs.

- AutoSuggest Client
- Computer Vision Client
- Custom Image Search Client
- Custom Search Client
- Custom Vision Prediction Client
- Custom Vision Training Client
- Entity Search Client
- Image Search Client
- Local Search Client
- News Search Client
- Partial Graph API
- Spell Check Client
- Video Search Client
- Visual Search Client
- Web Search Client

sgcOpenAPI Azure APIs

Find below a list of the currently available APIs.

- API Client
- ACE Provisioning ManagementPartner
- ADHybridHealthService
- AdvisorManagementClient
- Anomaly Detector Client
- Anomaly Finder Client
- ApiManagementClient
- AppConfigurationManagementClient
- Application Insights Data Plane
- ApplicationClient
- ApplicationInsightsManagementClient
- AppPlatformManagementClient
- AppServiceCertificateOrders API Client
- AppServiceEnvironments API Client
- AppServicePlans API Client
- Artifact
- AttestationClient
- AuthorizationManagementClient
- AutomationManagement
- AutomationManagementClient
- Azure Action Groups
- Azure Activity Log Alerts
- Azure Addons Resource Provider
- Azure Alerts Management Service Resource Provider
- Azure Bot Service
- Azure CDN WebApplicationFirewallManagement
- Azure Data Catalog Resource Provider
- Azure Data Lake Storage
- Azure Data Migration Service Resource Provider
- Azure Dedicated HSM Resource Provider
- Azure DevOps
- Azure Enterprise Knowledge Graph Service
- Azure IoT Central
- Azure Location Based Services Resource Provider
- Azure Log Analytics
- Azure Log Analytics - Operations Management
- Azure Log Analytics Query Packs
- Azure Machine Learning Datastore Management Client
- Azure Machine Learning Model Management Service
- Azure Machine Learning Workspaces
- Azure Maps Resource Provider
- Azure Media Services
- Azure Metrics
- Azure Migrate Hub
- Azure Migrate V2
- Azure ML Commitment Plans Management Client
- Azure ML Web Services Management Client
- Azure Monitor Private Link Scopes
- Azure Reservation
- Azure Resource Graph
- Azure Resource Graph Query
- Azure SQL Database
- Azure SQL Database API spec
- Azure SQL Database Backup
- Azure SQL Database Backup Long Term Retention Policy
- Azure SQL Database Datamasking Policies and Rules
- Azure SQL Database disaster recovery configurations
- Azure SQL Database Import/Export spec
- Azure SQL Database replication links
- Azure SQL Server API spec
- Azure SQL Server Backup Long Term Retention Vault

- Azure Stack Azure Bridge Client
- azureactivedirectory
- AzureAnalysisServices
- AzureBridgeAdminClient
- AzureDataManagementClient
- AzureDeploymentManager
- AzureDigitalTwinsManagementClient
- AzureStack Azure Bridge Client
- BackupManagementClient
- BatchAI
- BatchManagement
- BatchService
- BillingManagementClient
- BlockchainManagementClient
- BlueprintClient
- CdnManagementClient
- CertificateRegistrationProvider API Client
- Certificates API Client
- CognitiveServicesManagementClient
- CommerceManagementClient
- Compute Admin Client
- ComputeDiskAdminManagementClient
- ComputeManagementClient
- ComputeManagementConvenienceClient
- Computer Vision
- ConsumptionManagementClient
- ContainerInstanceManagementClient
- ContainerRegistryManagementClient
- ContainerServiceClient
- Content Moderator Client
- Cosmos DB
- CostManagementClient
- Customer Lockbox
- CustomerInsightsManagementClient
- customproviders
- Database Threat Detection Policy APIs
- DataBoxEdgeManagementClient
- DataBoxManagementClient
- DatabricksClient
- DataFactoryManagementClient
- DataLakeAnalyticsAccountManagementClient
- DataLakeAnalyticsCatalogManagementClient
- DataLakeAnalyticsJobManagementClient
- DataLakeStoreAccountManagementClient
- DataLakeStoreFileSystemManagementClient
- DataShareManagementClient
- DeletedWebApps API Client
- DeploymentAdminClient
- DeploymentScriptsClient
- DeviceServices
- DevSpacesManagement
- DevTestLabsClient
- Diagnostics API Client
- DiskResourceProviderClient
- DnsManagementClient
- Domain Services Resource Provider
- DomainRegistrationProvider API Client
- Domains API Client
- Dynamics Telemetry
- Engagement.ManagementClient
- EngagementFabric
- EventGridManagementClient
- EventHub2018PreviewManagementClient
- EventHubManagementClient
- Execution Service

- ExpressRouteCrossConnection REST APIs
- FabricAdminClient
- Face Client
- FeatureClient
- Form Recognizer Client
- FrontDoorManagementClient
- GalleryManagementClient
- Guest Diagnostic Settings
- Guest Diagnostic Settings Association
- GuestConfiguration
- HanaManagementClient
- HDInsightJobManagementClient
- HDInsightManagementClient
- HealthcareApisClient
- HybridComputeManagementClient
- HybridDataManagementClient
- HyperDrive
- InfrastructureInsightsManagementClient
- Ink Recognizer Client
- InstanceMetadataClient
- IntuneResourceManagementClient
- iotDpsClient
- iotHubClient
- IoTSpacesClient
- KeyVaultClient
- KeyVaultManagementClient
- KustoManagementClient
- LogicAppsManagementClient
- LogicManagementClient
- LUIS Authoring Client
- LUIS Programmatic
- Machine Learning Compute Management Client
- Machine Learning Workspaces Management Client
- MaintenanceManagementClient
- ManagedLabsClient
- ManagedNetworkManagementClient
- ManagedServiceIdentityClient
- ManagedServicesClient
- Management Groups
- ManagementLinkClient
- ManagementLockClient
- MariaDBManagementClient
- Marketplace RP Service
- MarketplaceOrdering.Agreements
- MediaServicesManagementClient
- Microsoft Insights
- Microsoft NetApp
- Microsoft Storage Sync
- Microsoft.ResourceHealth
- Microsoft.Support
- MicrosoftSerialConsoleClient
- Mixed Reality
- ML Team Account Management Client
- MonitorManagementClient
- MySQLManagementClient
- NetworkAdminManagementClient
- NetworkExperiments
- NetworkManagementClient
- NotificationHubsManagementClient
- PeeringManagementClient
- Personalizer Client
- PolicyClient
- PolicyEventsClient
- PolicyMetadataClient
- PolicyStatesClient

- PolicyTrackedResourcesClient
- portal
- PostgreSQLManagementClient
- Power BI Embedded Management Client
- PowerBIDedicated
- PrivateDnsManagementClient
- Provider API Client
- QnAMaker Client
- QnAMaker Runtime Client
- Recommendations API Client
- RecoveryServicesBackupClient
- RecoveryServicesClient
- RedisManagementClient
- Relay
- RemediationsClient
- ResourceHealthMetadata API Client
- ResourceManagementClient
- Run History APIs
- RunCommandsClient
- SchedulerManagementClient
- SeaBreezeManagementClient
- SearchIndexClient
- SearchManagementClient
- SearchServiceClient
- Security Center
- Security Insights
- ServerManagement
- Service Fabric Client APIs
- Service Map
- ServiceBusManagementClient
- ServiceFabricManagementClient
- SharedImageGalleryServiceClient
- SignalRManagementClient
- SiteRecoveryManagementClient
- Software Plan RP
- SqlManagementClient
- SqlVirtualMachineManagementClient
- Storage Cache Mgmt Client
- StorageImportExport
- StorageManagementClient
- StorSimple8000SeriesManagementClient
- StorSimpleManagementClient
- StreamAnalyticsManagementClient
- SubscriptionClient
- SubscriptionDefinitionsClient
- SubscriptionsManagementClient
- Text Analytics Client
- TimeSeriesInsightsClient
- TopLevelDomains API Client
- TrafficManagerManagementClient
- Update Management
- UpdateAdminClient
- UsageManagementClient
- VirtualMachineImageTemplate
- VirtualWANAsAServiceManagementClient
- Visual Studio Projects Resource Provider Client
- Visual Studio Resource Provider Client
- VM Insights Onboarding
- VMwareCloudSimple
- WebApplicationFirewallManagement
- WebApps API Client
- WebSite Management Client
- windowsesu
- WorkbookClient
- Workload Monitor

OpenAPI Microsoft | Tenant

To build apps that use the Microsoft identity platform for identity and access management, you need access to an Azure Active Directory (Azure AD) *tenant*. It's in the Azure AD tenant that you register and manage your apps, configure their access to data in Microsoft 365 and other web APIs, and enable features like Conditional Access.

A tenant represents an organization. It's a dedicated instance of Azure AD that an organization or app developer receives at the beginning of a relationship with Microsoft. That relationship could start with signing up for Azure, Microsoft Intune, or Microsoft 365, for example.

Each Azure AD tenant is distinct and separate from other Azure AD tenants. It has its own representation of work and school identities, consumer identities (if it's an Azure AD B2C tenant), and app registrations. An app registration inside your tenant can allow authentications only from accounts within your tenant or all tenants.

Use an existing Azure AD tenant

To check the tenant:

1. Sign in to the Azure Portal. Use the account you'll use to manage your application.
2. Check the upper-right corner. If you have a tenant, you'll automatically be signed in. You see the tenant name directly under your account name.

If you don't have a tenant associated with your account, you'll see a GUID under your account name. You won't be able to do actions like registering apps until you create an Azure AD tenant.

Create a new Azure AD tenant

You'll provide the following information to create your new tenant:

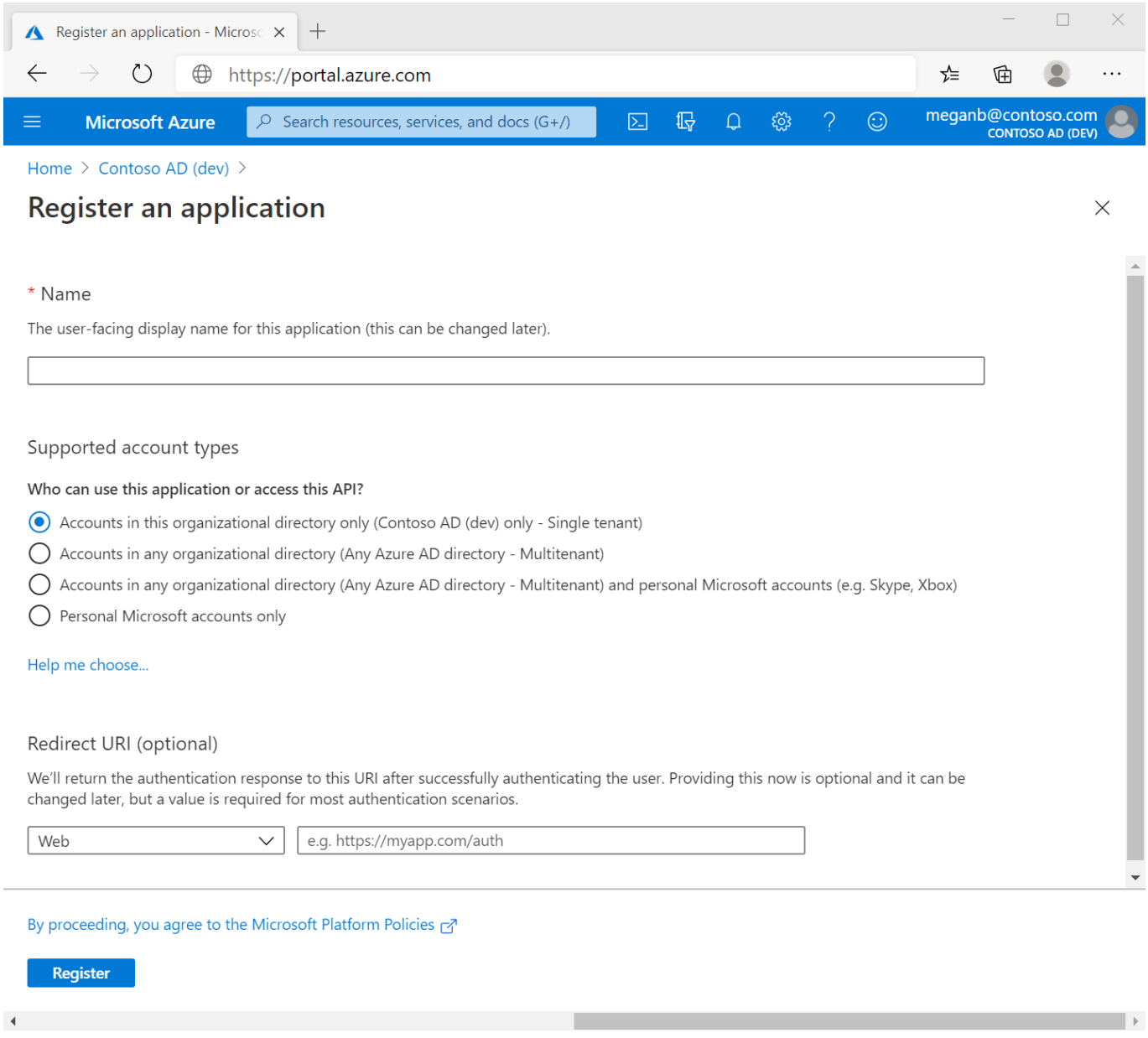
- **Organization name**
- **Initial domain** - Initial domain `<domainname>.onmicrosoft.com` can't be edited or deleted. You can add a customized domain name later.
- **Country or region**

OpenAPI Microsoft | Register Application

Registering your application establishes a trust relationship between your app and the Microsoft identity platform. The trust is unidirectional: your app trusts the Microsoft identity platform, and not the other way around.

Follow these steps to create the app registration:

1. Sign in to the Azure Portal.
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations** > **New registration**.
5. Enter a display **Name** for your application. Users of your application might see the display name when they use the app, for example during sign-in. You can change the display name at any time and multiple app registrations can share the same name. The app registration's automatically generated Application (client) ID, not its display name, uniquely identifies your app within the identity platform.
6. Specify who can use the application, sometimes called its *sign-in audience*.
7. Select **Register** to complete the initial app registration



Add a redirect URI

A *redirect URI* is the location where the Microsoft identity platform redirects a user's client and sends security tokens after authentication.

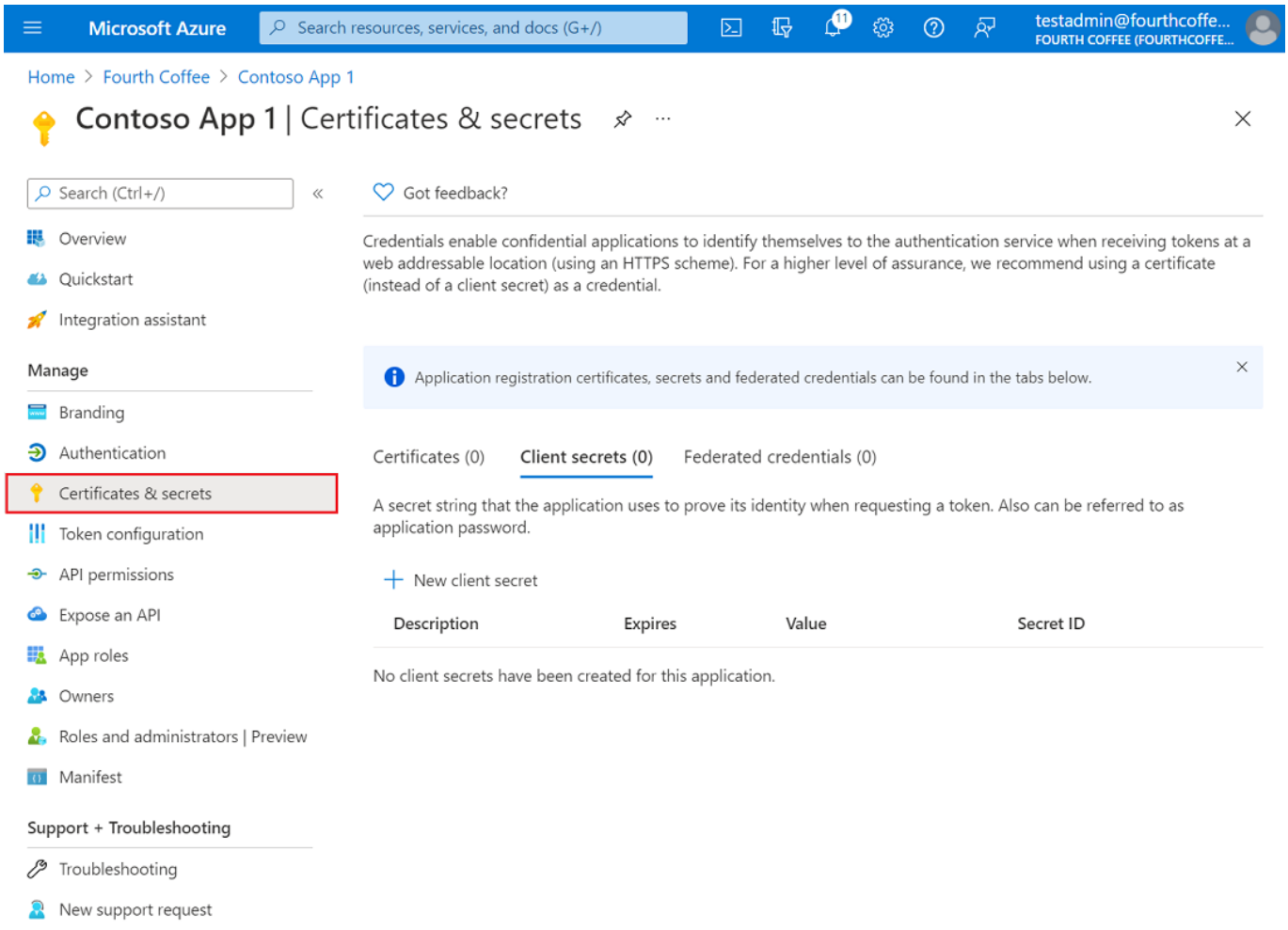
In a production web application, for example, the redirect URI is often a public endpoint where your app is running, like `https://contoso.com/auth-response`. During development, it's common to also add the endpoint where you run your app locally, like `https://127.0.0.1/auth-response` or `http://localhost/auth-response`.

This **RedirectURI** will be used later to configure the **sgcOpenAPI Microsoft Client**.

Add credentials

Credentials are used by confidential client applications that access a web API. Examples of confidential clients are web apps, other web APIs, or service-type and daemon-type applications. Credentials allow your application to authenticate as itself, requiring no interaction from a user at runtime.

You can add both certificates and client secrets (a string) as credentials to your confidential client app registration.



Add a client secret

Sometimes called an *application password*, a client secret is a string value your app can use in place of a certificate to identify itself.

Client secrets are considered less secure than certificate credentials. Application developers sometimes use client secrets during local app development because of their ease of use. However, you should use certificate credentials for any of your applications that are running in production.

1. In the Azure portal, in **App registrations**, select your application.
2. Select **Certificates & secrets** > **Client secrets** > **New client secret**.
3. Add a description for your client secret.
4. Select an expiration for the secret or specify a custom lifetime.
    - Client secret lifetime is limited to two years (24 months) or less. You can't specify a custom lifetime longer than 24 months.
    - Microsoft recommends that you set an expiration value of less than 12 months.
5. Select **Add**.
6. *Record the secret's value* for use in your client application code. This secret value is *never displayed again* after you leave this page.

OpenAPI Microsoft | OAuth2 Code

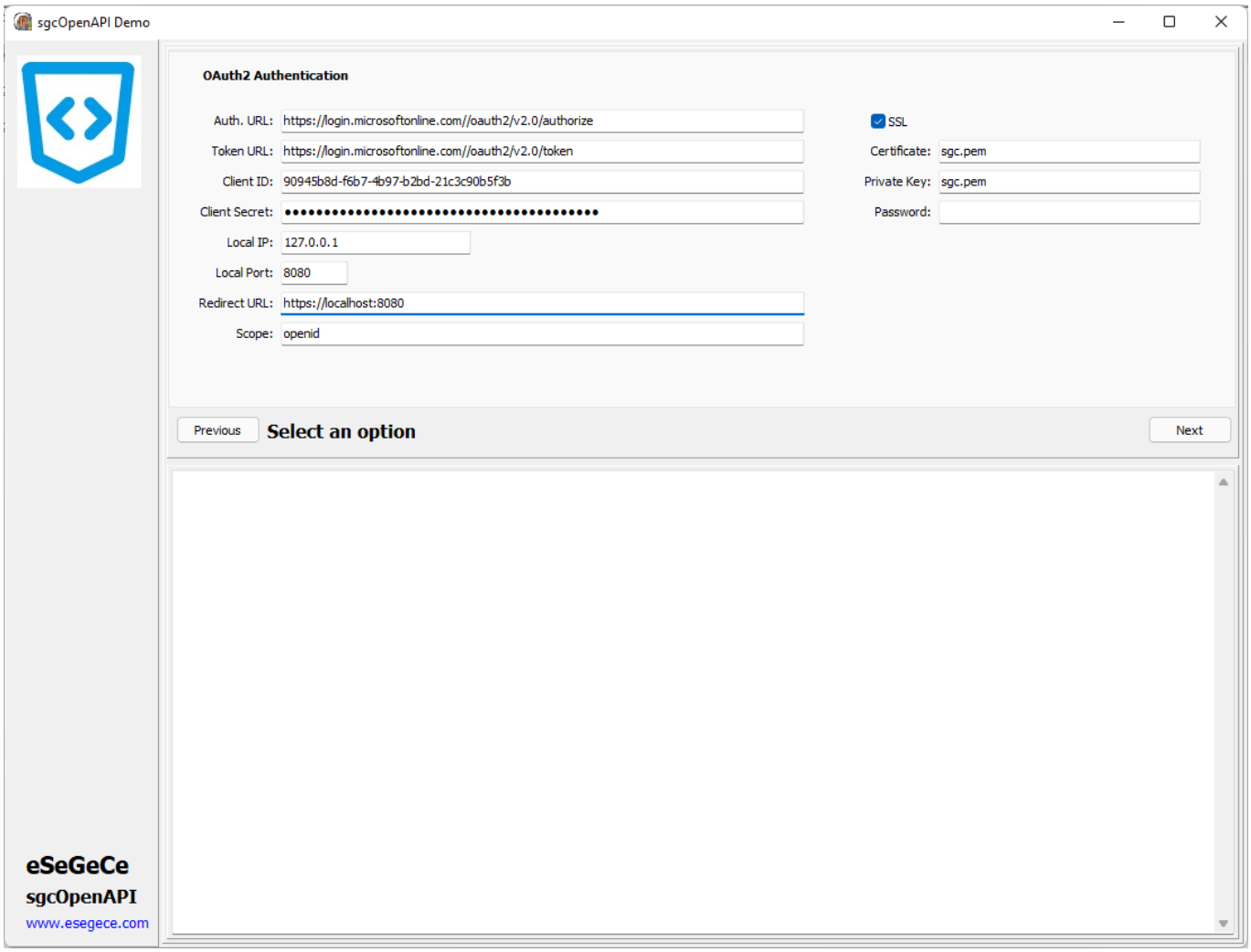
Using OAuth2 Code Grant Flow requires interaction with the user to login and get the required privileges.

Once you have the Tenant Id and the Credentials, you can configure the OAuth2 properties for Code Grant.

To configure the OpenAPI Client for OAuth2 Code Grant, configure the property **MicrosoftOptions.Authentication** with the following value:

```
GetOpenAPIClient->MicrosoftOptions->Authentication = oamaOAuth2Code;
```

Then you can configure the OAuth2 parameters




```
GetOpenAPIClient->MicrosoftOptions->Authentication = oamaOAuth2Code;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->AuthURL = "https://login.microsoftonline.cc
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->TokenURL = "https://login.microsoftonline.c
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientId = "90945b8d-f6b7-4b97-b2bd-21c3c90b5f3b";
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientSecret = "client_secret";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->IP = "127.0.0.1";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->Port = 8080;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSL = True;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->Port = 8080;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->CertFile = "sgc.pem";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->KeyFile = "sgc.pem";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->Password = "";
```

```
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->RedirectURL = "https://localhost:8080";
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->Scope->Text = "

openid

";
```

The Tenant ID must be configured for the Authentication and Token URLs, just replace the correct Tenant Id in the url.

Microsoft only allows the URL **localhost** if you are listening in the Local IP, so set the **redirect URL** with localhost as dns name instead of configuring with the IP address.

The **scope** value depends on the API, check the Microsoft / Azure documentation for every API.

The **first time** a request is made, it shows the web-browser asking the user to login to his Microsoft account. If the user has already logged in previously, it will make the HTTP request directly.

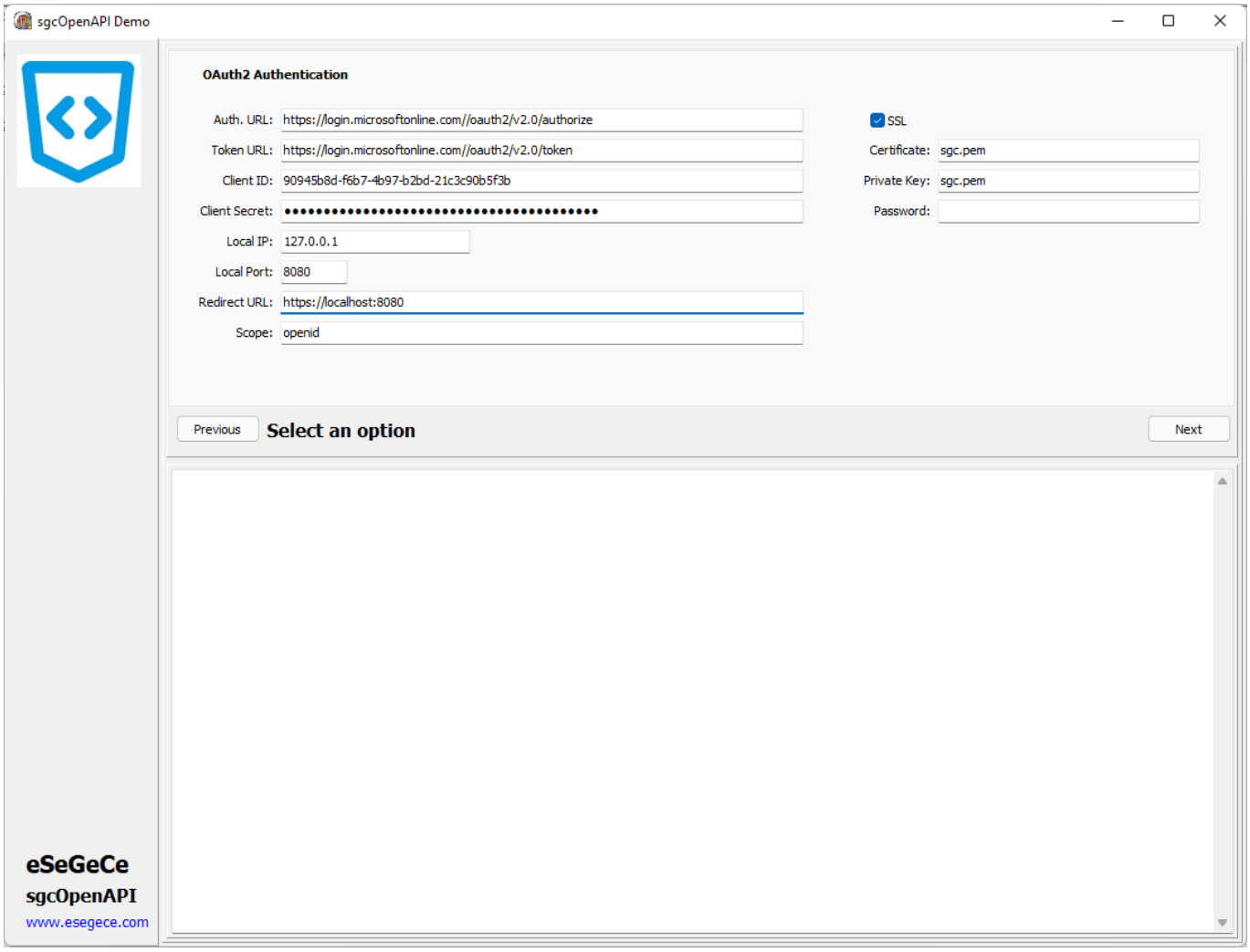
OpenAPI Microsoft | OAuth2 Credentials

Using OAuth2 Code Grant Flow doesn't require interaction with the user, so is suitable for services, daemons... or any application that must run without user interaction.

Once you have the Tenant Id and the Credentials, you can configure the OAuth2 properties for Code Grant.

To configure the OpenAPI Client for OAuth2 Code Grant, configure the property **MicrosoftOptions.Authentication** with the following value:

```
GetOpenAPIClient->MicrosoftOptions->Authentication = oamaOAuth2Credentials;
```




```
GetOpenAPIClient->MicrosoftOptions->Authentication = oamaOAuth2Credentials;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->AuthURL = "https://login.microsoftonline.co
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->TokenURL = "https://login.microsoftonline.c
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientId = "90945b8d-f6b7-4b97-b2bd-21c3c90b5f3b";
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientSecret = "client_secret";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->IP = "127.0.0.1";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->Port = 8080;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSL = True;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->Port = 8080;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->CertFile = "sgc.pem";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->KeyFile = "sgc.pem";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->Password = "";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->RedirectURL = "https://localhost:8080";
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->Scope->Text = "
```

```
https://graph.microsoft.com/.default
";
```

The Tenant ID must be configured for the Authentication and Token URLs, just replace the correct Tenant Id in the url.

Microsoft only allows the URL **localhost** if you are listening in the Local IP, so set the **redirect URL** with localhost as dns name instead of configuring with the IP address.

The **scope** value depends on the API, check the Microsoft / Azure documentation for every API.

OAuth2 credentials don't require any user interaction, so no browser will be opened the first HTTP request call.

OpenAPI Microsoft | Graph

Microsoft Graph exposes REST APIs and client libraries to access data on the following Microsoft cloud services:

- Microsoft 365 core services: Bookings, Calendar, Delve, Excel, Microsoft 365 compliance eDiscovery, Microsoft Search, OneDrive, OneNote, Outlook/Exchange, People (Outlook contacts), Planner, SharePoint, Teams, To Do, Workplace Analytics
- Enterprise Mobility + Security services: Advanced Threat Analytics, Advanced Threat Protection, Azure Active Directory, Identity Manager, and Intune
- Windows services: activities, devices, notifications, Universal Print
- Dynamics 365 Business Central

Get Current User

```
GetOpenAPIClient->MicrosoftOptions->Authentication = oamaOAuth2Code;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->AuthURL = "https://login.microsoftonline.co
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->TokenURL = "https://login.microsoftonline.c
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientId = "90945b8d-f6b7-4b97-b2bd-21c3c90b5f3b";
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientSecret = "client_secret";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->IP = "127.0.0.1";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->Port = 8080;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSL = True;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->Port = 8080;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->CertFile = "sgc.pem";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->KeyFile = "sgc.pem";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->Password = "";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->RedirectURL = "https://localhost:8080";
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->Scope->Text = "

openid

";
GetOpenAPIClient->meuserGetUser();
```

APIs

The following APIs have been generated using the eSeGeCe sgcOpenAPI Generator and provided for free to any user. The source code of the interface is provided with the trial, so you can see what you can expect if you purchase a license of any of the private apis like Google, Amazon or Microsoft.

API	Description
Geolocation	The Abstract IP Geolocation API takes an IP address and translates it into a loca tails, such as an address, timezone, and more.

OpenAPI | AbstractApi Geolocation

What is the IP Geolocation API?

The Abstract IP Geolocation API takes an IP address and translates it into a location, as well as many other details, such as an address, timezone, and more.

What are some use cases for the IP Geolocation API?

There are many powerful use cases for IP geolocation API's and data. These include but are not limited to:

- Automatically redirect users to relevant sites or sub sites based on their location
- Automatically detect and displaying a user's location, country, or timezone without requiring them to explicitly make this customization
- Customize the content or experience of a website or app based on the user's location. E.g., showing a user's local weather, tax and VAT rates, currency, news, public holidays, etc.
- Filter out users based on their location, e.g., if you're unable to offer your services to users in a particular country.
- Requiring that a user accepts certain terms as required by local regulations, such as GDPR cookie banners for European Union citizens

Where get an API Key?

Just register in abstractapi.com and you will get an api key for free

https://www.abstractapi.com/api/ip-geolocation-api

Sample Code

The following code returns the info about the IP Address provided

```
ShowMessage(GetOpenAPIClient->Retrieve_the_location_of_an_IP_address("asdfkjlkj32i3j2liwj3es", "88.5.12.4"));
```

HTML Components

sgcHTML is a server-side HTML/UI component framework that lets you build complete web interfaces from Delphi, C++ Builder and .NET. Every component renders to Bootstrap 5 markup and stays interactive through htmx.

Introduction

The HTML Components are a family of more than 60 visual components (charts, grids, forms, dashboards, navigation, overlays, chat and authentication widgets). You configure each component through ordinary published properties and methods, then read its **HTML** property to obtain clean, Bootstrap 5 markup ready to serve over an HTTP or WebSocket server.

Every component descends from **TsgcHTMLComponent** and overrides the `GetHTML` method to emit its markup. Pages are served by a **TsgcHTMLEngine_Server** (or an existing **TsgcWebSocketHTTPServer**), and a **TsgcHTMX_Engine_Server** adds htmx so user interactions round-trip back to your Pascal or C# code and the server swaps HTML fragments in place — with no hand-written JavaScript.

How It Works

1. Drop one or more HTML components on a form (or create them at runtime).

2. Set their properties, optionally bind a dataset, and add items or data.

3. Read the component's **HTML** property and serve it, or wrap it in a **TsgcHTMLTemplate_Bootstrap** page.

4. Attach a **TsgcHTMX_Engine_Server** for live, server-driven updates over WebSockets.

Steps 1 and 2 can also be done visually instead of in code, dropping the same components on a form and composing the page with the Design Surface — see Runtime vs Design-Time for the full comparison.

Component Families

Family	Components
Navigation & Structure	NavBar, Sidebar, Breadcrumb, Tabs, Pagination, Toolbar, TreeView, Stepper, Dropdown, ButtonGroup, ListGroup, DashboardLayout
Data & Tables	Grid, DataTable, Calendar, Scheduler, Timeline, KanbanBoard, Gantt
Charts & Visualization	Chart, Gauge, Diagram, Map
Forms & Inputs	Form, Edit, Memo, CheckBox, RadioGroup, Select, InputGroup, AutoComplete, DatePicker, FileUpload, RichEditor, Rating
Content & Layout	Panel, StatCard, Accordion, Carousel, Image, Avatar, Video

Overlays & Feedback	Modal, Offcanvas, Popover, Toast, Snackbar, Notification, Spinner, Placeholder
Chat & AI	ChatBox, Chat, AIChat
Authentication	Login, SocialLogin, OAuthCallback, WebAuthnLogin

QuickStart

Six small demos that teach sgcHTML one step at a time. Each one is a Delphi form with a handful of components, and each one runs as a real web page with no HTML to write.

Introduction

Who It Is For

The QuickStart is for Delphi developers who know forms, the Object Inspector and event handlers, and who want a web front end for their application without learning HTML, JavaScript or request handling first. If you can build a VCL window, you can follow every demo in this series.

The Zero-Code Idea

You drop sgcHTML components on an ordinary Delphi form and set their properties in the Object Inspector, exactly as you would build a desktop window. A TsgcHTMLDesignServer connects that form to a sgcWebSockets HTTP server, so the form itself becomes the web page, with no HTML, no routes and no request handler to write. When the user clicks a button or submits a form in the browser, an **OnExecute** event handler runs your Delphi code on the server, and only the component that changed is redrawn in the page.

This is the design-time way of building a page. The same components can also be created and filled in code, see Runtime vs Design-Time.

The Six Demos

The demos are in the folder **Demos\60.HTML\00.QuickStart**. Open them in order: each one adds a single new idea to the previous one, and each folder has a readme.html that explains it.

Demo	What it teaches
01.HelloWorld	The smallest sgcHTML application: four components on a Delphi form become a styled web page, with no HTML, no routes and no request handler.
02.Components	A small dashboard made of the main components (navigation bar, stat cards, chart, panel, progress bar, input controls and tabs), all configured in the Object Inspector.
03.Events	The web version of Button1Click: a button clicked in the browser runs an event handler on the server, and only the component it changed is updated in the page.
04.Forms	A web form posts its values to the server, and one Delphi event handler reads them, checks them and writes the answer into a card, without reloading the page.
05.Database	List, add, edit and delete rows: a grid bound to a FireDAC query through a TDataSource, three small forms and one short SQL statement per form.

06.Pages	A small web site of three pages. Each page is a Delphi form, the menu is a frame dropped on every form, and all the pages share one HTTP server.

Hello World

The first demo needs four components on a form, all of them set in the Object Inspector:

Component	Class	Properties
WSServer	TsgcWebSocketHTTPServer	Port = 8081. The web server that answers the browser.
DesignServer	TsgcHTMLDesignServer	Server = WSServer, Surface = Surface, PagePath = '/'. Sends the page built from this form when a browser asks for PagePath.
Surface	TsgcHTMLDesignSurface	Title = 'Hello sgcHTML'. The page itself: its title, its theme and the sgcHTML components dropped on the form.
Panel1	TsgcHTMLComponent_Panel	Title = 'Hello World', Body = 'Welcome to sgcHTML...'. The only thing on the page, a Bootstrap card.

The only line of code starts the server:

Run the application and open http://127.0.0.1:8081/ in any browser on the same machine. The panel is served as a styled web page, together with the Bootstrap files it needs.

Getting Started Guide

The same path, explained step by step with screenshots, is available as a PDF: sgcHTML Getting Started.

See Also

- TsgcHTMLDesignServer
- Runtime vs Design-Time
- Designing in the Browser
- HTML Components

Runtime vs Design-Time

sgcHTML pages can be built two ways — in code at runtime, or visually at design time with the Design Surface. Both use the same components and produce the same HTML; the difference is only how you author the page.

Two Ways to Build a Page

Runtime

This is the approach described in HTML Components: create or drop the components, set their properties and bind data in Pascal, C# or .NET code, then read the resulting **HTML** property. It has always been the primary way to build a page, and remains fully supported — nothing about it changes with the Design Surface.

Design-Time (Design Surface)

The same components can instead be dropped directly on an ordinary VCL form at design time and arranged visually — a component's position and width on the form infer its Bootstrap row and column, and its published properties are tuned in the Object Inspector instead of in code. A **TsgcHTMLDesignSurface** component added to the form reads that layout: call `sgcHTMLDesignBuildFromControls(Surface, Self)` once, typically at startup, to walk the form's dropped components into the surface's node tree, then `Surface.GetBodyHTML` (after `Surface.Invalidate`) to render it.

For markup the component palette does not cover, `Surface.AddHTML('', aParentNodeId)` adds a fixed HTML node into the same tree. Its **Order** property controls where it falls among the form's components — a negative value places it above everything else — and setting **Slot** to `'Body'` places it inside a dropped component's own body or container, found with `Surface.Nodes.FindByComponent(aComponent)`.

The same layout can also be edited live in a browser while the form stays open in the IDE — see Designing in the Browser.

Same Components, Same Output

The Design Surface only changes how a page's structure is expressed, not the components themselves or their behavior. A component dropped on a designed form is an ordinary component instance — `Grid1.LoadFromDataSet(qry)` works exactly as it would on a page built entirely in code. The two authoring styles can also be mixed freely on the same page.

When to Use Each

Approach	Best for

Runtime	Page structure that is itself dynamic or data-driven (loop-generated rows, conditional sections, content that depends on state only known at runtime); one source shared across Delphi 7–13 with no IDE dependency; developers who prefer expressing layout in code.
Design-Time	Laying out mostly-static page structure visually and seeing its shape while building it; tuning properties through the Object Inspector rather than in code; faster iteration on a page's visual composition.

Both produce components that compile and run identically once built — choosing one over the other is an authoring-time preference, not a difference in runtime capability.

Current Limitations

The Design Surface is under active development and its palette keeps growing. A few current gaps are worth knowing about:

- Some components' own wrapper markup — for example **TsgcHTMLComponent_Panel**'s `card`/`card-body` shell — has no property to suppress it. A page that must not be wrapped in a card should add its content as a root-level fixed node instead of inside a dropped Panel.
- **TsgcHTMLComponent_Form**'s **Fields** do not currently apply column classes or spans the way hand-written markup can; a form that needs specific per-field column widths may need some fields expressed as fixed markup instead.
- Not every property exists on every sibling component — **UseColorClass**, for instance, exists on **Panel** but not on **Login** or **StatCard**. Check the actual component before assuming a property carries across a family.
- **TsgcHTMLComponent_Grid** does not yet expose sortable column headers as a design-time property.

See It in Action

The **Demos\60.HTML** folder has the same set of demos built both ways, side by side: **01.RunTime** has the original code-built pages, and **02.DesignTime** has every one of them rebuilt entirely on forms with the Design Surface — AdminCRUD, LiveMonitor, Portal, HTMX, Grid, Site, Components, Helpdesk, ShopAssistant, WebBroker and WebBrokerHTML, alongside the ERP demo that started the pattern. Compare the two folders for the same demo to see both approaches produce the same page.

Designing in the Browser

The Design Surface can also be edited live, in an ordinary web browser, while the form stays open in the IDE — components added, moved or removed there are written straight back onto the form.

Two Ways to Open It

The Design in Browser Verb

Right-click a **TsgcHTMLDesignSurface** on the form and choose **Design in Browser...** from its context menu. This is available in every design package, Delphi 7 through 13, and opens the live canvas in the system's default browser.

The same context menu also has **Preview in Browser...**, which is a different, read-only thing: it writes a standalone snapshot of the page, filled with sample data, to a temporary file and opens that instead. Nothing is listening on the other end, so nothing dropped there goes anywhere — use it to show the page, use Design in Browser to change it.

The Web Design Tab (Delphi 10.4 and later)

On Delphi 10.4 and later, a form carrying a **TsgcHTMLDesignSurface** gets its own **Web Design** editor tab, alongside Code and Design. It embeds a live preview of the page and a small toolbar: **Refresh** re-reads the form and re-renders it (the tab does not watch the form for changes on its own); **Preview in browser** opens the same read-only snapshot as the verb above; **Design in browser** opens the live editable canvas; and a device-width selector renders the page at Desktop, Tablet or Phone width, so a narrow layout is an actual narrow viewport rather than the same page squeezed with CSS.

The embedded preview depends on the WebView2 Runtime, which is not guaranteed to start on every machine. Both browser buttons, and the right-click verb, sidestep that entirely — they open the system's own default browser and never depend on WebView2, so the live canvas is reachable even where the embedded preview is not.

The Form Is the Source of Truth

Every time the canvas is opened, from either entry point, it clears itself and rebuilds from whatever is on the form right now, rather than reopening a snapshot left over from an earlier session. This keeps the canvas honest, but it has one consequence worth knowing: a row or column added purely with the canvas's own layout tools, with no component behind it, is discarded the next time the canvas is reopened. A component dropped from the canvas is unaffected, because it is created on the form itself the moment it is added, so the next rebuild picks it up exactly like any component placed there in the Form Designer.

Edits Write Back to the Form

Adding, moving or removing a component in the browser changes the actual component sitting on the form, not a copy of it. The canvas is served over a small local listener, so a browser request naturally arrives on a background thread; the component creation this can involve is automatically marshalled onto the main thread, which is a VCL requirement components rely on internally — nothing about this needs any handling on your part. Editing also

marks the module modified, the same as any other design-time change, so the IDE's save prompt reflects what the canvas changed.

The Local Listener, and Exporting

The canvas is served by a small listener bound to **127.0.0.1** only, so it is reachable from the machine running the IDE and from nowhere else. That matters because the designer routes are not authenticated: they serve the source of the page being designed and they set published properties by name.

Exporting the designed page to real files is done through **ExportFolder** on the design host. The destination is chosen by you, in the Object Inspector or in code, and is never taken from the request. It is empty by default, and while it is empty an export writes nothing. Once it is set, an export writes `index.html`, the surface's CSS and JS files and `<SourceUnitName>.pas` into that folder, overwriting any existing files of those names.

Requirements

Entry point	Needs
Design in Browser... verb	Delphi 7–13, and any installed web browser.
Web Design tab	Delphi 10.4 or later. The embedded preview needs the WebView2 Runtime; its own Design in browser and Preview in browser buttons do not.

AI Page Generator

Describe a page in plain language and the designer builds it out of your own components.

Introduction

The AI page generator is the natural language side of the visual designer. A human types "an orders page with a search box, a grid of the orders of the month and a chart of the totals by customer", and **TsgcHTMLDesignAIGenerator** answers with a **design tree**: the same JSON the designer surface reads and writes.

The model never writes markup and never writes code. It picks component classes out of the registry and sets their published properties, and everything it answers goes through the surface's own strict reader before a single component exists. What comes out is an ordinary page of ordinary components, which the designer can then edit by hand and export as Object Pascal like any other.

It lives in the unit **sgcHTMLDesign_AI**. The provider path needs the AI units, which are Windows only: without them the component still compiles and answers with one line saying there is no provider, so a project that uses it builds on every platform.

Quick Start

1. In the IDE, open the **Web Design** tab of a form. The prompt box sits in the toolbar next to Generate and Refine. Configure the provider once and it is read again every time the tab opens.

2. In an application that hosts the designer itself, create the generator, point it at the surface and hand it to the host:

```
FGenerator = new TsgcHTMLDesignAIGenerator(this);
FGenerator->Surface = oSurface;
FGenerator->APIKey = ReadSetting("openai.key");
FGenerator->Model = "gpt-4o-mini";

oHost = new TsgcHTMLDesignHost(this);
oHost->Surface = oSurface;
oHost->Generator = FGenerator;
oHost->Enabled = true;
```

3. Or drive it in code, with no designer at all:

The Design Tree

A design tree is what the surface saves and reads: **SaveTreeJSON** writes it and **LoadTreeJSON** reads it back. Five node kinds, nested by their children:

kind	What it is	What it carries
section	A titled block of the page	caption, css

row	A Bootstrap row	css
column	A Bootstrap column	width (1 to 12), offset
component	One of your components	class, props, slot
html	A piece of text	html

```
{"version":1,"nodes":[
  {"kind":"section","caption":"Orders","children":[
    {"kind":"row","children":[
      {"kind":"column","width":12,"children":[
        {"kind":"component","class":"TsgcHTMLComponent_Grid",
         "props":{"ElementID":"orders","Title":"Orders of the month"}}
      ]}]}]}]}
```

The properties are written as text, exactly as the property inspector of the designer reads and writes them, so whatever the inspector can set is what a tree can carry. A collection, a nested object and an event are not text and do not travel.

Why a Generated Page Is Safe

Three boundaries, and none of them is the prompt:

- **The schema is built from the registry.** The request carries a JSON schema whose class enum is exactly the set of registered components, narrowed further by the **Classes** property, so the model cannot name a class that does not exist or that you did not offer.
- **The answer is normalized. NormalizeTree** drops every key it does not know, drops a node whose class is not registered, caps the tree at **MaxNodes** and rewrites the property list into the shape the surface reads.
- **The surface reads strictly. LoadTreeJSON** skips and REPORTS an unknown node kind, an unregistered class, a property that does not exist or is not writable, and a value the property refuses. It answers False only when the JSON itself could not be read, so a caller always reads the error list to know what survived.

Markup is the one thing a generated tree is never trusted with: an `html` node is loaded as **escaped text** unless **AllowRawHTML** is switched on, so a model that was talked into answering a script puts that script on the page as visible text and nothing else.

Nothing of the page being designed is sent to the provider except what **Refine** needs: the current tree, which is the page itself. No dataset row, no connection string and no property of a component that is not in the tree ever travels.

From a Picture

A page can also be designed from an image: a screenshot of the application being replaced, a mockup out of a design tool, a photograph of a sketch on paper. **GenerateFromImage** takes a file name, **GenerateFromImageBase64** takes an image a host already holds, and both answer the same design tree **Generate** answers, through the same normalization.

```
vTree = FGenerator->GenerateFromImage("C:\\shots\\orders.png",
  "ignore the sidebar", oErrors);
if (vTree != "")
  FGenerator->Apply(vTree, oErrors);
```

The description is optional and narrows what to look at rather than describing the page. The picture is read by the model, so this needs a provider and a model that read images; one that does not answers nothing and says so in the error list.

In the designer the button is **From image**, next to Generate and Refine. It opens a file dialog, the browser reads the picture and posts it to **/ai/image**, and what comes back is the same candidate with the same Accept and Discard. The picture travels to the provider and nowhere else, and what it produces is a tree of registered components exactly like any other: an image cannot smuggle markup onto the page any more than a sentence can.

Generate and Refine

Generate starts from nothing: the description, the layout rules and the palette. **Refine** starts from the page as it is now and asks for the WHOLE page after the change, never a patch, so the candidate can be compared with the current page before anything is applied.

Neither of them touches the page. They answer a tree, and it is **Apply** that puts it on the surface. That is what lets the designer show a candidate side by side and keep Accept and Discard honest.

In the Designer

The browser hosted designer adds a prompt box and two buttons to its toolbar, and the candidate opens as a full preview with **Accept** and **Discard**. Accepting replaces the page and keeps one step of undo. The routes behind it, under the designer base path:

Route	What it does
/ai/generate?prompt=...	Asks for a page and keeps the answer as a candidate. Nothing is created.
/ai/refine?prompt=...	The same, starting from the current page.
/ai/image (POST)	A page designed from a picture. The body carries `image` (base64, with or without the data: prefix), `mime` and an optional `prompt`. It is a POST because a screenshot does not fit in a query string.
/ai/preview	The candidate rendered on a surface of its own, so the page being designed is untouched.
/ai/accept	The candidate becomes the page. The components of the previous one are freed and one undo is kept.
/ai/undo	Back to the page as it was before the last accept.
/ai/discard	The candidate is forgotten.
/ai/status	Whether a generator and a provider are configured, which is what disables the buttons.

The designer is a developer tool and its listener is bound to loopback, exactly as it was before. A generator on a production server would be a hole for the same reason the designer itself is: leave **Enabled** off there.

Where the Keys Live

In the IDE the provider settings belong to the developer's machine, not to the project being designed: **sgcHTMLDesign_IDE_Options** keeps them in the registry under the current user, and the API key is stored obfuscated rather than in clear. They are read again every time the Web Design tab opens, so a key typed in the settings is picked up without restarting the IDE.

The page that edits them is **Tools > Options > Third Party > sgcHTML AI page generator**: the provider, the API key, the model, a base URL for a provider that is not at its usual address, the language the captions are asked in, and the largest tree that will be accepted.

Nothing is written into the .dfm, the .dproj or the page, which is what keeps an API key out of source control. An application that hosts the designer itself sets **APIKey** from wherever it keeps its own secrets.

What Comes Back Is Yours

An accepted page is made of ordinary components owned by the form, so the Object Inspector edits them, the designer exports them and **GetUnitSource** rebuilds them as Object Pascal that compiles with the designer uninstalled. There is no runtime dependency on the generator, and a page generated once never asks a provider again.

MCP Apps

Your Delphi page inside Claude, ChatGPT and VS Code, with the components you already use.

Introduction

An MCP App is a tool whose result the host does not only read: it RENDERS a page next to the answer. The tool says which resource carries its user interface, the host reads that resource, puts the HTML in a sandboxed iframe and talks to it over postMessage.

TsgcHTMLComponent_MCPApp is the sgcHTML side of that. It registers the tool and the resource on a TsgcAI_MCP_Server, answers the resource with a page built out of ordinary sgcHTML components, and routes what the page asks back into the application. It lives in the unit **sgcHTML_MCPApp**.

Editions. MCP needs the **SGC_AI_MCP** define, which reaches the Enterprise and All-Access editions and the sgcAI pack.

Quick Start

1. Add an app: the tool the model calls, what it does, and the page.

2. Answer **OnToolCall** with what the model should read and, optionally, the JSON the page should read.

3. Answer **OnFragment** with the markup a control of the page asked for, which is what makes paging, sorting and submitting work inside the host.

What Gets Declared

RegisterApps declares three things per app on the server:

What	Name	Why
The tool	`ToolName`	What the model calls. Its `_meta.ui.resourceUri` tells the host which resource carries the page, and `preferredSize` what to size the frame to.
The fragment tool	`<ToolName>.fragment`	What the PAGE calls, never the model: it answers the markup of one fragment.
The resource	`ui://sgchtml/<tool-name>`	The document itself, served as `text/html;profile=mcp-app` .

The component subscribes to the tool and the resource events of the server, so a server that already serves tools of its own keeps serving them: a request that belongs to no app is left alone. An application that handles those

events itself calls **ProcessToolsCall** and **ProcessResourcesRead** instead, which answer `False` when the request was not an app's.

The Document the Host Renders

The iframe a host renders is sandboxed and usually has no network of its own, so nothing may be linked. With **InlineAssets** on, which is the default, the stylesheet, the script and the page travel in ONE document. Switch it off only for a host on the same machine that can serve the assets.

Every document carries the bridge, **window.sgcMCPApp**:

- it sends `ui/initialize` as soon as the page is ready and applies the theme the host answers with to `data-bs-theme`;
- it reports the height on every change, because a host sizes the frame from what the page says;
- it shows what `ui/notifications/tool-result` carries in the element with the id `sgc-mcp-result`, when the page has one;
- it registers the htmx extension `mcp`, which the body carries as `hx-ext="mcp"`.

That extension is what makes the page work with no HTTP server behind it: every `hx-get` and `hx-post` becomes a `tools/call` of the fragment tool, whose text result is swapped in exactly as an HTTP answer would have been. A grid page, a sort, a filter and a form submit therefore work unchanged inside the host.

What the Model Reads, and What the Page Reads

A tool result carries both: **aText** is the summary the model reads and reasons about, and **aStructured** the JSON the page reads. Answer both when they differ, which they usually do: the model wants a sentence and the page wants numbers.

The component keeps NO state of its own. What a tool call means is the application's to decide, exactly as it decides what a route means in a web application, and the MCP session the request arrived on is what identifies who is asking.

The Demo

Demos\60.HTML\01.RunTime\19.MCPApp is a console server on port 5725 that publishes two apps over one MCP endpoint, and ships a **reference host** so they can be seen without installing anything. Open `http://localhost:5725/`, pick a tool, and the left panel does what a real host does: it calls the tool, reads the resource it points at, renders it in a frame with `sandbox="allow-scripts"`, answers `ui/initialize`, sizes the frame from what the page reports and forwards the page's `tools/call` messages. The postMessage log under it is the conversation between the two.

The endpoint a real host connects to is `http://localhost:5725/mcp`, served by **TsgcWSAPIServer_MCP**. Its **MCPServer** property is what the component is given:

```
FMCP = new TsgcWSAPIServer_MCP(NULL);
FMCP->Server = FHTTP;
FMCP->EndpointOptions->Endpoint = "/mcp";

FApps = new TsgcHTMLComponent_MCPApp(NULL);
FApps->MCPServer = FMCP->MCPServer;
FApps->OnRender = DoRender;
FApps->OnToolCall = DoToolCall;
FApps->OnFragment = DoFragment;
FApps->RegisterApps();
```

The inline assets come out of the packed resources, so an application that publishes MCP Apps links **sgcHTMLResources.RES** and uses **sgcHTMX_Resources**. Without them the document would link a stylesheet the iframe cannot fetch.

Linking at design time

Instead of assigning **MCPServer** in code, set the **MCPEndpoint** property of the MCP Apps component to the MCP endpoint component (`TsgcWSAPIServer_MCP`) in the Object Inspector. When the endpoint is linked, the component uses the endpoint's MCP server and registers its apps automatically (after the form is loaded); linking another endpoint or clearing the property unregisters them first. **RegisterApps** can be called more than once: apps that are already registered are skipped. In .NET, set **MCPEndpoint** in code; the endpoint exposes its **MCPServer** property publicly.

```
MCPApps->MCPEndpoint = MCPAPI;
```

CRUD Pages

The list, the form and the write path of one table, as one component.

Introduction

A CRUD page is always the same page: a list with a search box and paging, a form to create and to edit, a confirmation to delete, and the code that writes. **TsgcHTMLComponent_CRUD** owns all of it. It composes the grid and the form through their public properties, and it writes through an **adapter**, so the same page works over a TDataSet, over your own SQL, or over anything else your application can answer.

It lives in the unit **sgcHTML_Component_CRUD** and renders like every other sgcHTML component: markup out, actions in.

Quick Start

With no columns declared the component reads them off the dataset: the caption from `DisplayLabel`, the required flag from `Required`, the length from `Size`, and the key field is shown but never edited. Declare **Columns** to choose which fields appear, where, and how.

The Five Things It Does

Method	Action posted	What comes back
ShowList(search, page)	crudList	The list, with the toolbar and the pager.
ShowNew	crudNew	An empty form.
ShowEdit(key)	crudEdit	The form of one record, or the list when the key locates nothing.
Save(key, values, html)	crudSave	True when it was written. When it was not, the markup is the form again with the errors in it.
DeleteRecord(key, html)	crudDelete	True when it was deleted, and the list either way.

ProcessAction routes all five, and answers `False` for anything else so a page with several components can go on looking. The paths the routes of an engine use are built by **RoutePath**, so the page and the host can never disagree about them: `/customers`, `/customers/new`, `/customers/{id}/edit`, `/customers/save`, `/customers/{id}/delete`.

The Adapter Is the Write Path

Everything that reads a page of rows or changes a record goes through a **TsgcHTMLCRUDAdapter**: `Locate`, `Read`, `List`, `Insert`, `Update` and `Delete`. The default one drives the bound dataset. An application that runs its own SQL either handles **OnApplyUpdates**, which is asked before the adapter and can answer the write itself, or gives the component an adapter of its own with **SetAdapter**.

The list contract is one line per row, the values of the visible columns separated by tabs, with **the key as the last cell**. The key is never rendered as a column: it is what the row actions carry.

Validation

Before a write the component checks what it knows: a required column that is empty and a value longer than the column takes are refused, and **OnValidate** is asked for everything else. A refusal is not an exception: the form comes back with the errors above it, and nothing was written.

What the adapter refuses is reported the same way, so a unique constraint or a foreign key from the database reads as a message on the form rather than as a stack trace.

Security

- **Every key arrives from the browser**, so nothing is edited or deleted before the key has been LOCATED through the adapter. That is what stops one user reaching another user's record by changing a number.
- **The actions are a set.** A page whose Actions do not include caDelete refuses a delete even when a forged form posts one, and renders no delete control.
- **Values are data.** Every cell and every field goes through the escaping node layer, so a record that holds markup is shown as text. The row actions are the only markup the component emits into a cell, and it emits them itself.
- **The CSRF rule and the roles are the engine's.** Declare the routes with `RequireLogin` and `RequireRoles`, and the engine refuses what should not reach the component at all.

How a Page Talks Back

Every control of the page posts the same way: the search box, the New button, the Edit and Delete of a row and the form itself are forms marked `data-sgc-ws-send`, and each one carries three hidden fields, `action` (crudList, crudNew, crudEdit, crudSave or crudDelete), `crud` (which component asked) and, where it means something, `key`. The host reads them and calls **ProcessAction**, which answers the markup to swap in.

A message sent over the socket carries no path, so it never reaches the router and never gets the router's own check. Authorizing it is the host's job, and **TsgcHTMX_Engine_Server.MessageSession** is what that takes: it answers the session behind the handshake cookies of the connection, or nil when the handshake was not same origin or no session matches.

The Demo

The Customers area of **Demos\60.HTML\01.RunTime\02.AdminCRUD** is the same page written twice. `/customers` is the component with an adapter of six methods over the application's database pool; `/customers-classic` is the hand written page it replaced, four request handlers and about four hundred lines of page builders, still there to read side by side.

Common Properties

Every sgcHTML component descends from TsgcHTMLComponent and shares four properties for the id, CSS classes, inline style and extra attributes of its outer element.

Introduction

All sgcHTML visual components (charts, grids, forms, navigation, overlays and the rest) descend from **TsgcHTMLComponent**. Besides their own specific properties, every one of them exposes the four common properties below. They apply to the outer element the component renders, so you can give any component a stable id, add Bootstrap or custom CSS classes, set an inline style or attach extra HTML attributes without changing the component itself.

Rather than repeating these four rows in every component topic, they are documented here once. Each component reference lists only the properties specific to that component; assume these four are available in addition.

Common Properties

Property	Type	Default	Description
ElementID	string	''	Value of the id attribute of the component's outer element. Use it so scripts and CSS can target the element, or to reference it from htmx hx-target.
CSSClass	string	''	Extra CSS class (or space-separated list of classes) added to the outer element, alongside the classes the component already emits.
Style	string	''	Inline CSS written to the style attribute of the outer element.
Attributes	string	''	Extra raw HTML attributes (for example data-* attributes or htmx hx-* attributes) appended once at the end of the component's root opening tag. The value is written as it is, not encoded, so escape any quotes inside attribute values yourself ("). When empty, nothing is added. For components that render several top-level elements it goes on the element that carries the component id: the `<table>` for Grid, PivotTable, TreeGrid and UserManagement, the `<html>` element for Site, the tab list for horizontal Tabs.

Example

```
oCard->ElementID  = "revenue-card";
oCard->CSSClass   = "shadow-sm border-0";
oCard->Style      = "max-width:24rem";
oCard->Attributes = "hx-get=\"/refresh\" hx-trigger=\"every 5s\"";
```

Node / Composition Layer

The node layer is a set of lightweight, renderable building blocks you compose in code to produce Bootstrap 5 markup, without writing HTML strings by hand.

Introduction

Alongside the high-level components, sgcHTML ships a low-level composition layer. A node is a lightweight renderable object (it descends from TsgcHTMLNode, not from TComponent) that emits a piece of HTML through its read-only **HTML** property. You build a tree of nodes in code and read the root node's HTML, which is cleaner and safer than concatenating strings. The same layer powers TsgcHTMLPageBuilder and the document templates.

The generic building block is **TsgcHTMLContainer**, an element (a by default) that holds child nodes. On top of it, ready-made Bootstrap widgets such as modals, accordions and toasts are provided as nodes with rich slots.

Memory: a node list owns the nodes you add to it, so free only the root node of a tree. Never free a node you have already added to another node or list.

Node Classes

Class	Description
TsgcHTMLContainer	Generic element (default) that holds child nodes. Set the Void property to emit a void element (such as <hr> or) with no closing tag.
TsgcHTMLImage	An <img> element with source, alt text and styling.
TsgcHTMLModal	Bootstrap 5 modal dialog with Header, Body and Footer slots.
TsgcHTMLOffcanvas	Bootstrap 5 offcanvas side panel with a Body slot and placement.
TsgcHTMLAccordion	Bootstrap accordion of collapsible sections.
TsgcHTMLCarousel	Bootstrap carousel / slideshow of slides.
TsgcHTMLToast	Bootstrap toast notification.
TsgcHTMLProgress	Bootstrap progress bar.
TsgcHTMLSpinner	Bootstrap loading spinner.
TsgcHTMLListGroupNode	Bootstrap list group of items.

TsgcHTMLInput-Group	Bootstrap input group (an input with add-ons).
TsgcHTMLButton-GroupNode	Bootstrap button group.
TsgcHTMLRadio	A radio-button input.
TsgcHTMLBread-crumb	Bootstrap breadcrumb trail.
TsgcHTML-NavbarNode	Bootstrap navigation bar.

Output Encoding Helpers

Component properties are plain values, not markup: the component escapes them for the place they are written into, exactly once. The helpers that do it live in **sgcHTML_Helpers** and are public, so your own nodes and raw markup can use the same ones.

Function	Use it for
sgcHTMLEncode	Text content. Escapes &, <, >, the double quote and the apostrophe.
sgcHTMLAttrEn-code	An attribute value. Escapes the same five characters.
sgcHTMLSani-tizeURL	A URL in href or src. An unsafe scheme such as javascript: becomes #; the optional second argument allows data:image/... URIs.
sgcJSEncode	A value inside a JavaScript string literal. It also neutralizes a closing script tag.
sgcHTMLSafeID	An element id. Keeps only letters, digits, `_`, `-`, `:` and `.`, so the same value can be written into an id, a selector and a script.
sgcHTMLSafe-Name	A tag or attribute name. Returns the value when it matches a letter followed by letters, digits or hyphens, and the default you pass otherwise.
sgcHTMLCSS-Value	A CSS value. Removes the angle brackets, the curly braces, the semicolon, the backslash and the control characters, so the value cannot end the declaration, the rule or the style element. Inside an attribute, pass the result through sgcHTMLAttrEncode as well.

In the node layer, **AddText** encodes what you give it and **AddRaw** writes it as it is, so use AddRaw only for markup you build yourself.

Example: build an element tree

TsgcHTMLContainer holds children added with Add, AddText and AddRaw. Read the HTML property of the root to render the whole tree.

Example: a Bootstrap modal

Widget nodes expose slots (node lists) you fill in, then render through the HTML property.

Localization

sgcHTML ships built-in translations for the fixed texts its components render (button captions, field labels, day names, status messages). Select one with sgcHTMLSetLanguage. Right-to-left languages also switch the page layout automatically.

Introduction

Many sgcHTML components include fixed texts: the Close caption of a modal, the Previous / Next of a carousel, the Search box of a data table, the day-name headers of a calendar, the Sign In label of the login form and so on. These strings come from a shared **TsgcHTMLLocale** object returned by **sgcHTMLDefaultLocale**.

Call **sgcHTMLSetLanguage** once at start-up to load one of the built-in translations into the default locale. Every component then renders its fixed texts in that language. For wording that is not covered, or to change a single string, set the individual properties of the locale directly.

sgcHTMLSetLanguage

Loads the built-in translation for the given language into the default locale (sgcHTMLDefaultLocale).

```
void __fastcall sgcHTMLSetLanguage(TsgcHTMLLanguage aLang);
```

```
// render component texts in Spanish
sgcHTMLSetLanguage(htmlLangSpanish);

// customise a single string afterwards
sgcHTMLDefaultLocale()->SearchText = "Filtrar";
```

TsgcHTMLLanguage

The languages with a built-in translation:

Value	Language
htmlLangEnglish	English
htmlLangSpanish	Spanish
htmlLangFrench	French
htmlLangGerman	German
htmlLangPortuguese	Portuguese

htmlLangItalian	Italian
htmlLangDutch	Dutch
htmlLangSwedish	Swedish
htmlLangNorwegian	Norwegian
htmlLangPolish	Polish
htmlLangCzech	Czech
htmlLangTurkish	Turkish
htmlLangRussian	Russian
htmlLangChinese	Chinese
htmlLangJapanese	Japanese
htmlLangKorean	Korean
htmlLangArabic	Arabic (right-to-left; the pages switch to a right-to-left layout automatically, see below)

Right-to-left layout

A page is rendered right-to-left when its language reads from right to left. sgcHTML decides this automatically, so you do not need to change your pages:

- After **sgcHTMLSetLanguage(htmlLangArabic)** the default locale has **Direction** set to 'rtl', and every page rendered afterwards is right-to-left. Any other language resets it to an empty string.
- The page language alone is also enough: when **HtmlLang** (or **Lang** on the Site component) is Arabic, Hebrew, Persian, Urdu, Pashto, Yiddish, Dhivehi, Uyghur, Central Kurdish or Sindhi, with or without a region such as 'ar-EG' or 'he-IL', the page is right-to-left even when no built-in translation exists for that language.

On a right-to-left page the html tag gets **dir="rtl"**, and Bootstrap's right-to-left stylesheet is loaded instead of the normal one: the default **/bootstrap.min.css** becomes **/bootstrap.rtl.min.css** (embedded in the library and served like the normal one), and the CDN stylesheet switches to its CDN right-to-left copy. A stylesheet path of your own is left unchanged. The components mirror as well: the site sidebar and icon rail, steppers, timeline, chat, presence, tree view, slider, splitter, grid column resizing and frozen columns, notifications, avatars and the kanban board. Charts, gauges, instruments and the Gantt time axis keep their geometry on purpose. Left-to-right pages render exactly as before.

To force a direction, set **HtmlDir** on TsgcHTMLTemplate_Bootstrap, TsgcHTMLPageBuilder and TsgcHTMLComponent_Login, or **Dir** on TsgcHTMLComponent_Site, to 'rtl' or 'ltr'. Leave it empty for the automatic choice.

```
// Arabic texts and a right-to-left layout
sgcHTMLSetLanguage(htmlLangArabic);

// Hebrew: right-to-left from the language code alone
Site->Lang = "he";

// Arabic texts, but keep the layout left-to-right
Site->Lang = "ar";
Site->Dir = "ltr";
```

The same decision is available to your own code: **sgcHTMLGetDirection(aHtmlDir, aHtmlLang)** returns the direction a page gets ('rtl', 'ltr' or an empty string for the default left-to-right), and **sgcHTMLIsRTLLanguage(aLang)** tells whether a language code is a right-to-left one.

Customising strings

TsgcHTMLLocale holds one property per translatable text, grouped by component (pagination, modal, carousel, data table, file upload, calendar day names, login, OAuth callback, WebAuthn, spinner, video, the navigation texts **ReconnectTitle**, **ReconnectText**, **ReloadText** and **RequestFailedText** used by the reconnect dialog and the error toast of the page template, the nine texts of the AI Data Pack (**AIInterpretedAs**, **AISuggest**, **AIAccept**, **AIReject**, **AIDiscard**, **AIAsk**, **AIFilled**, **AIPasteHint** and **AIPasteButton**), and a set of general captions such as Submit, Cancel, Save and Delete). The data grid also uses **GridPageOfText** (the "of" of the pager text, 'of'), **GridSelectedText** (the bulk actions bar, '%d selected'), and the inline editor validation messages **GridRequiredText** ('Required'), **GridMinValueText** ('Value must be at least %s'), **GridMaxValueText** ('Value must be at most %s') and **GridInvalidFormatText** ('Invalid format'); the pivot table uses **PivotFieldsText** ('Fields'), **PivotRowsText** ('Rows'), **PivotColumnsText** ('Columns') and **PivotValuesText** ('Values'), the stepper input uses **DecreaseText** ('Decrease') and **IncreaseText** ('Increase') for its decrease and increase buttons, and the presence component uses **PresenceOnline** ('Online'), **PresenceAway** ('Away'), **PresenceBusy** ('Busy') and **PresenceOffline** ('Offline') for its status labels. **Direction** is 'rtl' for a right-to-left language and empty otherwise. Read the shared instance through sgcHTMLDefaultLocale and set any property to override the built-in wording. Each built-in language also has its own routine (sgcHTMLLocaleSpanish, sgcHTMLLocaleGerman and so on) that fills a locale you supply, in case you keep your own instance.

Routing

How TsgcHTMX_Router maps request paths and HTTP methods to Delphi handlers: literal and template paths, method filters, precedence, and transport-neutral dispatch.

Introduction

A TsgcHTMX_Router holds a collection of routes. Each TsgcHTMX_Route has a **Path**, an optional set of **Methods**, optional access rules (**RequireLogin**, **RequireRoles**) and an **OnRoute** handler that returns the HTML to send back. Assign the router to the Router property of TsgcHTMX_Engine_Server, of the WebBroker engine or of the http.sys engine, and the engine runs the matching route for HTTP requests and for htmx WebSocket messages.

Quick Start

```
oRouter = new TsgcHTMX_Router(this);

TsgcHTMX_Route *oRoute = oRouter->Routes->Add();
oRoute->Path = "/customers";
oRoute->OnRoute = CustomersRoute;      // GET

oRoute = oRouter->Routes->Add();
oRoute->Path = "/customers/{id}";
oRoute->Methods = TsgcHTMXRouteMethods() << hrmPut << hrmDelete;
oRoute->OnRoute = CustomerChangeRoute;

oHTMX->Router = oRouter;
```

The Route Handler

OnRoute is a `TsgcHTMXRouteEvent = procedure(Sender: TObject; const aRequest: TsgcHTMXRequest; var aResponse: string) of object`. The request object has these members:

Member	Type	Description
Path	string	Decoded request path.
Method	string	HTTP method as received. WebSocket messages use the method member of the message, POST when it is missing.
ConnectionGuid	string	Guid of the WebSocket connection of a message. Empty for HTTP requests.
QueryParams	TStringList	Query parameters as name=value lines. The Indy and WebBroker hosts also merge the fields of a url-encoded form body here. For a WebSocket message, every scalar member of the JSON message.
Params	TStringList	Values of the `{name}` segments of a template route, decoded.

Headers	TStringList	Request headers as name=value lines. Empty for WebSocket messages.
Cookies	TStringList	Request cookies as name=value lines. Empty for WebSocket messages.
Body	string	Request body as text. Empty for GET, HEAD and multipart requests and for WebSocket messages.
Session	TsgcHTMLSession	Session of the request when the engine has an Auth, nil when the request is anonymous. Owned by the engine.
IsHTMX	Boolean	True when the request carries `HX-Request: true`, so it came from htmx.
IsBoosted	Boolean	True when it also carries `HX-Boosted: true`, so it is a boosted link or a boosted form.
ResponseHeaders	TStringList	Headers the route adds to its HTTP answer, as name=value lines (HX-Push-Url, HX-Trigger, HX-Redirect and the other htmx response headers). Names must be HTTP tokens, control characters are removed from the values and the framing headers are ignored. A route that returns no content but adds a valid header is still answered. Ignored for WebSocket messages.

A route can tell a boosted navigation, another htmx request and a plain request apart with IsBoosted and IsHTMX, and can drive the browser with the response headers, see Navigation and Live Updates.

Set aResponse to the HTML to send back. An empty response means not handled: the host falls through (a 404 or the next dispatcher) and no cookie is sent. The handler runs in a server thread, and an exception it raises is not caught by the engine.

Path Parameters

A route path is a template when at least one whole segment is written as `{name}`, for example `/customers/{id}` or `/orders/{order}/lines/{line}`. Braces that do not fill a whole segment, as in `/files/v{n}`, are literal text.

- The request must have the same number of segments: `/customers/{id}` matches neither `/customers` nor `/customers/7/edit`. A trailing slash adds an empty segment, so `/customers/` does not match it either.
- A parameter never matches an empty segment.
- The value is taken from the raw path and decoded once, UTF-8 aware. An encoded slash does not split segments, so `/files/a%2Fb` gives `name=a/b` for `/files/{name}`.
- Literal segments, and literal routes, are compared without case with the decoded request path.
- The values arrive in `aRequest.Params`, one name=value line per parameter.

Methods

Methods is a set of TsgcHTMXRouteMethod: hrmGet, hrmPost, hrmPut, hrmPatch and hrmDelete. A route with Methods answers exactly those methods; HEAD and OPTIONS never match it. An empty set, the default, keeps the behaviour routes had before Methods existed:

Where	Methods answered by a route with an empty Methods set
TsgcHTMX_Engine_Server (Indy) and the http.sys engine	GET only
TsgcHTMX_Engine_Server_WebBroker	Every method (they still never replace the built-in login and logout endpoints)
WebSocket messages	Any method
TsgcHTMX_Router.HandleRequest(aPath, aMethod, aGuid, aParams)	Any method

Use an explicit Methods set for every route that changes data. The built-in login and logout endpoints of Auth are only replaced by a route whose Methods include hrmPost (see Sessions, CSRF and Authentication).

Precedence

1. The root path, `/` or an empty path, always returns the page of the engine Template, whatever the method. A route registered for `/` is only reached by WebSocket messages.
2. Literal routes, in registration order: the first one whose Path equals the request path and that accepts the method wins.
3. Template routes, in registration order.
4. With an Auth, a POST to Auth.LoginPath or Auth.LogoutPath is the exception: it goes to the built-in endpoints unless the matching route lists hrmPost in its Methods. A route with an empty Methods set never replaces them.

So `/customers/new` is found before `/customers/{id}` even when the template was added first, and two templates that match the same path are resolved by registration order. Before the router, the engine serves the PWA manifest and service worker (when enabled) and its built-in js and css files.

Access Rules

RequireLogin and **RequireRoles** are applied by the engine, not by the router. An anonymous HTTP request is redirected to the sign-in page, a session without the role gets 403, and an engine without Auth never runs such a route. See Sessions, CSRF and Authentication.

WebSocket Messages

An htmx WebSocket message is a JSON object. The engine takes the route path from its `path` member or, when it is missing, from `HEADERS.HX-Trigger-Name` or `HEADERS.HX-Trigger`. The method comes from the `method` member, POST when it is missing, and every scalar member (strings, numbers, booleans and nulls, path and method included) becomes a QueryParams line. Text that is not a JSON object is not routed. OnHTMXMessage runs first; the router is only asked when the event leaves the response empty, and a route response is pushed back to the connection that sent the message.

```
{"path": "/customers/42", "method": "DELETE", "reason": "duplicate"}
```

Transport-Neutral Dispatch

DispatchDirect serves a TsgcHTMXDispatchRequest into a TsgcHTMXDispatchResponse without any server. DispatchRequest (Indy), the WebBroker engine and the http.sys engine are thin adapters that copy their native request into it and write the response back, so a request behaves the same on every host, apart from the Methods rule above. Use it to put the engine behind a transport of your own, or to test routes.

```
TsgcHTMXDispatchRequest *oRequest = new TsgcHTMXDispatchRequest();
TsgcHTMXDispatchResponse *oResponse = new TsgcHTMXDispatchResponse();
try
{
  oRequest->Method = "DELETE";
  oRequest->Path = "/customers/42"; // raw path, as sent on the wire
  if (oHTMX->DispatchDirect(oRequest, oResponse))
    SendResponse(oResponse); // your own code
}
__finally
{
  delete oResponse;
  delete oRequest;
}
```

Path holds the raw, percent-encoded path. **Document** reads it decoded and, when assigned, stores a decoded path. Fill Path when you have the raw path, so that an encoded slash inside a parameter survives.

Asking Before Dispatching

CanDispatchRequest(aDocument) tells whether the engine would serve a GET for a decoded path, and CanDispatchRequest(aDocument, aMethod) answers for any method, the Auth login and logout endpoints included. Hosts that share a port with another dispatcher, such as the WebBroker bridges, use it to claim only their own requests.

The router itself offers FindRoute, HandleRoute and HandleRequest, and its Routes collection offers `FindRoute(aPath)` and `FindRoute(aPath, aMethod, aUnfiltered)`. The function `sgcHTMXDecodeFormValue` decodes one application/x-www-form-urlencoded name or value (a plus sign is a space and percent escapes are UTF-8).

PUT, PATCH and DELETE on the Indy Server

TsgcHTMX_Engine_Server hooks the OnCommandGet event of its server, which Indy fires for GET, HEAD and POST. Indy fires OnCommandOther for PUT, PATCH and DELETE, and the engine does not hook it. Forward it to DispatchRequest to reach those routes:

```
// TsgcHTMX_Engine_Server only hooks OnCommandGet (GET, HEAD, POST)
oServer->OnCommandOther = ServerCommandOther;

void __fastcall TForm1::ServerCommandOther(TIdContext* AContext,
  TIdHTTPRequestInfo* ARequestInfo, TIdHTTPResponseInfo* AResponseInfo)
{
  // PUT, PATCH and DELETE routes, with their gates and CSRF check
  oHTMX->DispatchRequest(AContext, ARequestInfo, AResponseInfo);
}
```

The WebBroker engine, the WebBroker bridges and the http.sys engine receive every method without this step.

See Also

- TsgcHTMX_Router
- TsgcHTMX_Engine_Server
- Sessions, CSRF and Authentication
- Navigation and Live Updates
- WebSocket or SSE
- Testing sgcHTML Pages

Sessions, CSRF and Authentication

How sgcHTML signs users in, keeps them in cookie sessions, protects routes by login and role, blocks cross-site request forgery and throttles password guessing.

Introduction

Three pieces work together. The unit **sgcHTML_Session** provides the session stores, TsgcHTMLSessionStore_Memory and TsgcHTMLSessionStore_File, and the TsgcHTMLSession class. The unit **sgcHTML_Auth** provides TsgcHTMLAuth. The HTMX engine, TsgcHTMX_Engine_Server, applies them to every request once its **Auth** property is assigned. The same Auth property exists on TsgcHTMX_Engine_Server_WebBroker and on the http.sys engine, and the WebBroker bridge engines inherit it.

Secure Your App in 10 Lines

A store, an Auth with a password check, and a flag on the routes to protect:

```
oStore = new TsgcHTMLSessionStore_Memory(this);
oStore->SweepInterval = 60;
oAuth = new TsgcHTMLAuth(this);
oAuth->SessionStore = oStore;
oAuth->OnAuthenticate = AuthAuthenticate; // checks user and password
oHTMX->Auth = oAuth;                      // TsgcHTMX_Engine_Server
TsgcHTMX_Route *oRoute = oRouter->Routes->Add();
oRoute->Path = "/orders";
oRoute->RequireLogin = true;
oRoute->OnRoute = OrdersRoute;
```

The password check and the sign-in page complete the setup. The page is an ordinary route for Auth.LoginPath. The POST of its form reaches the built-in login endpoint, because a route only replaces that endpoint when its Methods include hrmPost:

What the Engine Does per Request

With Auth assigned, TsgcHTMX_Engine_Server handles an HTTP request in these steps:

1. The PWA files and the built-in js and css assets are served first, without any session work. Requests the engine does not own get no session lookup and no cookie either.
2. For the page, a matching route or a POST to the login or logout path, the session is read with `Auth.CurrentSession`. That call checks the idle and absolute timeouts, touches the session, redeems a remember-me cookie when there is no session, and prepares the cookies to send: a new session, a rotated remember-me token, or the deletion of a stale cookie.
3. The session and the Auth become the request context of the thread (`sgcHTMLRequestSession` and `sgcHTMLRequestAuth`), so the components rendered during the request can use them, and the route receives the session as `TsgcHTMXRequest.Session`.
4. A route with RequireLogin or RequireRoles is checked first, then the CSRF token of a state-changing request.
5. The route runs, or the built-in endpoint answers. The page of Template at `/` is always served and is never gated.

6. The request context is restored, the session copy is freed, and the Set-Cookie headers are added to the response. A route that returns an empty string is not handled, so no cookie is sent. An exception raised by a route is not caught; the request context is restored before it propagates.

Route Gates

Two properties of TsgcHTMX_Route protect a route. **RequireLogin** admits any signed-in user. **RequireRoles** takes a comma separated list and admits a session that has at least one of those roles, compared without case; a value implies RequireLogin.

Situation	HTTP request	WebSocket message
The engine has no Auth	403 Forbidden, the route never runs	No response
Anonymous, Auth.LoginPath set	302 to `LoginPath?next=<path>`; htmx requests get 401 with an HX-Redirect header	No response
Anonymous, Auth.LoginPath empty	401 Unauthorized	No response
Session without any of the roles	403 Forbidden	No response
Session with the login or a role	The route runs	The route runs

`next` carries the requested path, percent-encoded, plus the query string for GET and HEAD requests only: the query of other methods can hold form fields, which must never end up in a URL. htmx requests get an HX-Redirect header instead of a 302 because htmx follows HX-Redirect with a full page load, while a 302 would be followed inside the fragment request.

Built-in Login and Logout

With an Auth, the engine answers two POST endpoints. A route only replaces one of them when it matches the path and its Methods include hrmPost; a route with an empty Methods set never does:

- **POST Auth.LoginPath** reads the url-encoded form fields `username`, `password`, `remember` (`on`, `1` or `true` ask for remember-me) and `next`. It calls `Auth.Authenticate`, which applies the login rate limit and fires OnAuthenticate. On success it calls `Auth.SignIn` with the id of the current session, so that id is replaced, and redirects to `next` when it is a safe relative path, or to Auth.AfterLoginPath. On failure it redirects to `LoginPath?error=1`, or to `LoginPath?error=locked` while the IP or the user name is locked, keeping `next`.
- **POST Auth.LogoutPath** checks the CSRF token of the session, calls `Auth.SignOut` and redirects to Auth.LoginPath, or to `/` when LoginPath is empty. An anonymous logout only redirects. A GET to the logout path is not an endpoint, so a link cannot sign anybody out.

The redirects are a 302 with a Location header for plain form posts, and a 200 with an HX-Redirect header for htmx requests.

The engine does not render the sign-in page: register a GET route for Auth.LoginPath that returns it. TsgcHTMLComponent_Login posts the field names above. With a request auth, and FormAction empty or left at its default `/login`, it posts to LoginPath and carries the CSRF token of the session. Set its NextURL from the `next` query parameter so the user returns to the page they asked for: the component only renders a relative path (one

leading `/`, not `//` or `/\`, no CR or LF), and the endpoint checks the value again and falls back to AfterLoginPath. Read the `error` query parameter to show the failure message.

CSRF Protection

Browsers send cookies with every request, including requests started by other sites. To stop another site from acting on behalf of a signed-in user, every session has a random CSRF token that state-changing requests must send back.

When `Auth.CSRFProtection` is True, the default, a request that has a session and a method other than GET, HEAD or OPTIONS gets 403 unless it carries the token of its session in one of these places, checked in this order:

1. the header named `Auth.CSRFHeaderName` (default `X-CSRF-Token`);
2. the field `Auth.CSRFFieldName` (default `csrf_token`) of a url-encoded form body;
3. the query parameter `Auth.CSRFFieldName`.

The token reaches the browser in three ways:

- **Meta tags.** When the page is rendered for a session, TsgcHTMLTemplate_HTMX adds `<meta name="csrf-token" content="...">` and `<meta name="csrf-header" content="X-CSRF-Token">` to the head. `sgcHTMX.min.js`, which the template includes, adds that header to every htmx request whose verb is not GET, HEAD or OPTIONS, so `hx-post`, `hx-put`, `hx-patch` and `hx-delete` need no extra markup.
- **Hidden fields.** TsgcHTMLComponent_Login and the logout form of TsgcHTMLComponent_Site carry a hidden field named CSRFFieldName. Add the same field to your own plain HTML forms, with the value of `Auth.IssueCSRF(session)`.
- **Your own scripts.** Read the csrf-token meta tag and send it in the header.

```
var token = document.querySelector("meta[name='csrf-token']").content;
fetch("/orders/7", { method: "DELETE", headers: { "X-CSRF-Token": token } });
```

The token is created at sign-in and does not change during the session. Anonymous requests and WebSocket messages are not checked (see Documented Limits). In handlers that the engine does not serve, check the token yourself with `Auth.ValidateCSRF`.

Remember Me

Set `Auth.RememberMeDays` to a number of days and handle `Auth.OnLoadUser`. A sign-in with the remember box checked then also writes a second cookie, named `Auth.RememberCookieName`, that lasts that many days.

- The cookie holds `selector.verifier`. The store keeps the selector, the user id, the expiry date and the SHA-256 of the verifier, never the verifier itself.
- When a request has no valid session but has that cookie, the token is consumed: it is deleted whatever happens next. When the verifier matches and the token has not expired, OnLoadUser is fired, and when it reports the user as found, a new session and a new token are issued. Both new cookies travel with the response.
- A known selector with a wrong verifier means the cookie was copied and one of the copies was already used. The token is revoked, so both the thief and the owner have to sign in again.
- Signing out revokes the token of the session, SignOutEverywhere revokes all the tokens of the user, and a rejected cookie is deleted from the browser.
- A remember-me cookie is never redeemed when `CurrentSession` is called without a cookie list (aSetCookies nil), which is what the engine does for WebSocket messages, because the new cookies could not reach the browser.

Login Rate Limiting

`Auth.Authenticate`, and therefore the built-in login endpoint, counts failed logins per client IP and per user name, compared without case. With the defaults of `Auth.LoginRateLimit`, 5 failures within 300 seconds lock that IP or that user name for 900 seconds. While a lock lasts even the right password is refused, OnAuthenticate is not fired and the login endpoint redirects with `error=locked`.

A successful login clears the counter of the user name only, so an attacker who owns one valid account cannot reset the counter of the IP used for a password spray. The counters live in memory, in each TsgcHTMLAuth instance: processes that share a TsgcHTMLSessionStore_File count separately, and a restart clears them. Behind a reverse proxy all requests may come from the address of the proxy, which then shares one IP counter; take that into account when you choose MaxAttempts. Set MaxAttempts to 0 to disable the limit.

Cookie Attributes

Property	Default	Effect
CookieName	'sgcsid'	Name of the session cookie.
RememberCookieName	'sgcrem'	Name of the remember-me cookie.
CookiePath	'/'	Path attribute. Set the mount path when the application lives under a sub path.
CookieDomain	''	Domain attribute. Empty writes a host-only cookie.
CookieSecure	True	Secure attribute: the cookies only travel over HTTPS. Browsers usually accept them on http://localhost too.
CookieHttpOnly	True	HttpOnly attribute: page scripts cannot read the cookies.
CookieSameSite	hssLax	SameSite attribute. hssNone always adds Secure.
AbsoluteTimeout	43200	Max-Age of the session cookie. 0 makes it a browser-session cookie.
RememberMeDays	0	Max-Age of the remember-me cookie, in days.

Every cookie is built with `TsgcHTMLAuth.BuildSetCookie`: each Max-Age comes with an explicit Expires date in RFC 1123 format, and a deleted cookie gets `Max-Age=0` and an Expires date in 1970. The engines write each cookie as its own raw Set-Cookie header. When a response carries two values for the same cookie, for example the deletion of a stale session followed by the session of a new login, only the last one is sent. Under WebBroker hosts other than the Indy bridge (ISAPI, Apache, CGI) the first cookie travels as a custom header and the others as response cookies.

Sessions and Timeouts

A session (TsgcHTMLSession) keeps the user id, the display name, the roles, free name=value Values, the CSRF token, the creation and last request times in UTC, the client IP and a hash of the User-Agent seen at sign-in. TsgcHTMLSessionStore_Memory lists all its members.

- **Idle timeout.** `Auth.IdleTimeout` (1800 seconds) ends a session that was not used for that long. Every request moves its LastSeen forward. TsgcHTMLSessionStore_File writes the new LastSeen at most once every TouchInterval seconds, which is then the precision of the idle timeout.
- **Absolute timeout.** `Auth.AbsoluteTimeout` (43200 seconds, 12 hours) ends a session that long after the sign-in, whatever the activity.
- **Sweeping.** An expired session is refused as soon as it is read, but it stays in the store until a sweep removes it. TsgcHTMLSessionStore_Memory can sweep in a background thread (SweepInterval); with TsgcHTMLSessionStore_File call Sweep yourself.
- **Ownership.** The stores and TsgcHTMLAuth hand out copies. A session returned by CurrentSession or SignIn belongs to the caller. The session in TsgcHTMXRequest.Session and in sgcHTMLRequestSession belongs to the engine and is freed when the request ends. To keep a change, call `Auth.SessionStore.Put`.
- **Binding.** A session is not tied to the IP or to the browser that created it. Compare RemoteIP or UserAgentHash yourself when your application needs that.

```
void __fastcall TForm1::ThemeRoute(TObject* Sender,
  TsgcHTMXRequest* const aRequest, UnicodeString &aResponse)
{
  if (aRequest->Session != NULL)
  {
    aRequest->Session->Values->Values["theme"] = aRequest->QueryParams->Values["theme"];
    FAuth->SessionStore->Put(aRequest->Session);
  }
  aResponse = "<span id=\"theme-saved\">Saved</span>";
}
```

Passwords

Never store plain passwords. `sgcHTMLPasswordHash` returns a PBKDF2-SHA256 hash with a random salt (210000 iterations by default) and `sgcHTMLPasswordVerify` checks a password against it in constant time. See TsgcHTMLAuth for the format.

WebSocket Messages

A message that the engine routes (DispatchMessage, when OnHTMXMessage leaves the response empty) runs with the session of the cookies the browser sent with the WebSocket handshake, provided that handshake came from the same origin. The route gates apply, but a denied message simply gets no response: there is no redirect over a socket.

- **Same-origin handshake.** A session is only resolved when the handshake had no Origin header, as with clients that are not browsers, or when the host and port of its Origin equal the Host header, compared without case and ignoring the ports 80 and 443. A handshake started by a page of another site, or with `Origin: null`, gives anonymous messages, whatever the SameSite attribute of the cookie, so another site cannot use the cookies of the user through a socket.
- Behind a reverse proxy the server must receive the public Host header (most proxies can preserve it); otherwise every browser handshake looks cross-site and all messages are anonymous.
- Only the session cookie is read. A remember-me cookie is never redeemed on a message, so a socket opened with an expired session stays anonymous until the page is loaded again over HTTP.
- An event stream of the SSE transport is bound to a session with the same rules, and the POST that carries its upstream messages is CSRF-checked like any other state-changing request, see WebSocket or SSE.
- Every message reads the session from the store again, so a sign-out, a timeout or SignOutEverywhere takes effect on the next message.
- The session is read from the handshake headers kept by the Indy and http.sys connections. On other connection types every message is anonymous.

- Messages are not CSRF-checked: the same-origin check of the handshake takes that role.
- OnHTMXMessage runs before the router, without the request session, and is not gated.

Using TsgcHTMLAuth without the HTMX Engine

TsgcHTMLAuth does not depend on the engine. The Admin CRUD demo (**Demos\60.HTML\01.RunTime\02.AdminCRUD**) serves its pages from its own Indy handlers and uses the component directly:

1. Once per request, before routing, call `CurrentSession` with the Cookie header, the client IP and the User-Agent, send the returned cookies, and publish the result with `sgcHTMLSetRequestAuth` and `sgcHTMLSetRequestSession` so the sgcHTML components see it. Clear both in a finally block, then free the session.
2. For the login POST, call `Authenticate`, then `SignIn` with the session id of the request (`GetCookieValue`) as aOldSessionID, and send the cookies. Use `IsLoginLocked` to report a lockout.
3. For every state-changing request, read the token from the CSRFHeaderName header or from the CSRFFieldName field and call `ValidateCSRF`. Accept the logout only as a POST with a valid token, then call `SignOut` and send the cookies.

The request context is a thread variable: always clear it before the session is freed, and never keep the pointer after the request.

Components That Use the Request Session

Component	Behaviour
TsgcHTMLTemplate_HTMX	Writes the csrf-token and csrf-header meta tags when the request has a session.
TsgcHTMLComponent_Login	With an Auth (its own or the request auth) and a FormAction that is empty or the default /login, posts to LoginPath; adds the CSRF hidden field when the request has a session, and a next field from NextURL.
TsgcHTMLComponent_Site	Shows the DisplayName of the session when Header.UserName is empty, and renders the header logout as a POST form to LogoutPath, with the CSRF token, when Header.LogoutHref is empty.
TsgcHTMLComponent_UserManagement, TsgcHTMLComponent_RolesPermissions	With RequiredRole set, render an alert (AccessDeniedText) and ignore messages unless the session has the role.
TsgcHTMLComponent_OAuthCallback	Shows the DisplayName of the session when UserName is empty.

These components read `sgcHTMLRequestSession` and `sgcHTMLRequestAuth`. Outside a request both are nil and the components render as they always did.

Hosts and HTTP Methods

Host	Requests that reach the engine
TsgcHTMX_Engine_Server on TsgcWebSock-etHTTPServer (Indy)	GET, HEAD and POST, through the OnCom-mandGet event that the engine hooks. PUT, PATCH and DELETE arrive in OnComman-dOther, which the engine does not hook: for-ward them as shown below.
TsgcHTMX_Engine_Server_WebBroker	Every method the WebModule dispatcher of-fers to it. Routes with an empty Methods set answer all of them.
TsgcHTMX_Engine_Server_WebBrokerBridge	GET requests it can serve, and any other method for which a route or the login and lo-gout endpoints answer (CanDispatchRequest). The bridge answers PUT, PATCH and DELETE it-self.
TsgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge	The same requests as the Indy bridge, an-swered in place.
TsgcHTMX_Engine_Server_HTTPAPI (http.sys)	Every method, through the OnHTTPRequest event of the server.

```
// TsgcHTMX_Engine_Server only hooks OnCommandGet (GET, HEAD, POST)
oServer->OnCommandOther = ServerCommandOther;

void __fastcall TForm1::ServerCommandOther(TIdContext* AContext,
  TIdHTTPRequestInfo* ARequestInfo, TIdHTTPResponseInfo* AResponseInfo)
{
  // PUT, PATCH and DELETE routes, with their gates and CSRF check
  oHTMX->DispatchRequest(AContext, ARequestInfo, AResponseInfo);
}
```

Documented Limits

- **Anonymous unsafe requests are not CSRF-checked.** There is no session to abuse, so a POST, PUT, PATCH or DELETE route without RequireLogin runs for anonymous visitors without a token. Gate such routes when they must not be public.
- **Anonymous login POSTs skip the CSRF check**, because the visitor has no token yet. A login POST made with a session must carry the token of that session.
- **RequireLogin and RequireRoles need an Auth.** On an engine without Auth such a route is never run: HTTP requests get 403 and WebSocket messages get no response.
- **PUT, PATCH and DELETE on the plain Indy server.** With TsgcHTMX_Engine_Server on TsgcWebSock-etHTTPServer, those routes are only reached when you forward OnCommandOther to DispatchRequest.
- **WebSocket messages are not CSRF-checked.** Instead, a message only runs with a session after a same-origin handshake, or after a handshake without an Origin header. Browsers always send Origin on a Web-Socket handshake, so a client without it is trusted like any HTTP client that holds the session cookie.
- **The page at `/` is not gated.** Keep private content in routes with RequireLogin.

- **Login throttling is per process.** The counters live in each TsgcHTMLAuth instance and are reset by a restart.

See Also

- TsgcHTMLAuth
- TsgcHTMLSessionStore_Memory
- TsgcHTMLSessionStore_File
- Routing
- Navigation and Live Updates
- WebSocket or SSE
- Testing sgcHTML Pages
- TsgcHTMX_Engine_Server

Testing sgcHTML Pages

Unit test helpers for server-rendered HTML: deterministic element ids, a small DOM with CSS selectors, HTML assertions, snapshot files, and request tests without a server.

Introduction

The unit **sgcHTML_Testing** contains what you need to test the markup that your pages and components produce. It depends on no test framework: every assertion raises `EsgcHTMLAssertion`, a descendant of Exception, with a message that says what was expected and what was found, and DUnit, DUnitX or a plain console runner report it as a failed test. Add the unit to your test projects only; the library does not use it.

Deterministic Element Ids

Many components generate element ids from a counter shared by the whole process (sgcHTMLUniqueID), so the same page gets different ids depending on what was rendered before it. Call `sgcHTMLSetIDSeed(aSeed)`, declared in sgcHTML_Helpers, at the start of a test: the next generated id is the prefix followed by aSeed + 1, the one after that aSeed + 2, and so on. `sgcHTMLSetIDSeed(0)` reproduces the ids of a freshly started process. When it is never called nothing changes.

The call is meant for test code only: calling it while a server is running can hand out an id that is already on a page.

```
sgcHTMLSetIDSeed(0);
TsgcHTMLComponent_Slider *oSlider = new TsgcHTMLComponent_Slider(NULL);
try
{
  vHTML = oSlider->HTML; // same ids on every run
}
__finally
{
  delete oSlider;
}
```

TsgcHTMLDOMLite

`TsgcHTMLDOMLite.Create(aHTML)` parses markup into an element tree that simple CSS selectors can query. The parser is tolerant, the way a browser is:

- comments, `<!DOCTYPE>`, CDATA sections and processing instructions are skipped;
- attributes may be double quoted, single quoted, unquoted or boolean (a boolean attribute has an empty value), and the first of two attributes with the same name wins;
- void elements (area, base, br, col, embed, hr, img, input, link, meta, source, track and wbr) and any element written as `<x/>` have no children;
- the content of `<script>` and `<style>` is kept as raw text, a `<` inside does not open a tag;
- an end tag with no open element is ignored, elements still open at the end of the input are closed there, and an li, dt, dd, option, tr, td, th or p start tag closes an open sibling of the same kind;
- tag and attribute names are lower-cased, while attribute values and text are kept exactly as written: entities are not decoded, so a test can check how a value was escaped.

Member	Description
TsgcHTMLDOMLite.Select(aSelector): TsgcHTML-DOMLiteElementList	Every matching element, in document order. The caller frees the list, never the elements.
TsgcHTMLDOMLite.SelectFirst(aSelector): Ts-gcHTMLDOMLiteElement	The first matching element, or nil.
TsgcHTMLDOMLite.Root	The document node: an empty TagName, no at-tributes and the top level elements as Children.
Element.TagName	Lower-cased tag name.
Element.Parent	Parent element, nil only for Root.
Element.Children	Child elements in document order; text is not list-ed.
Element.Text	Text of the element and of all its descendants, concatenated in document order. The content of script and style is included as raw text.
Element.Attr(aName)	Value of an attribute, an empty string when the el-ement does not have it.
Element.HasAttr(aName)	True when the element has the attribute.
Element.HasClass(aClass)	True when aClass is one of the space separated names of the class attribute, compared with case as in CSS.

The elements belong to the TsgcHTMLDOMLite that parsed them and are freed with it. Supported selectors:

Selector	Matches
`button`	Elements with that tag name (any case in the selector).
`*`	Any element.
`#save`	The element whose id is save.
`.btn`, `.btn.btn-primary`	Elements that have all the listed classes.
`[disabled]`, `[hx-on:click]`, `[@click]`	Elements that have the attribute. The name may contain : and @.
`[type=submit]`, `[type="submit"]`, `[type='submit']`	Elements whose attribute has exactly that value, as written.
`button.btn[type=submit]`	A compound selector: every condition on the same element.
`form .is-invalid`, `table tr td`	The descendant combinator (a space), at any depth.

Anything else, such as the child and sibling combinators (>, +, ~), pseudo classes or attribute operators like `[attr^=value]`, raises EsgcHTMLAssertion, so a test never silently matches the wrong elements.

Assertions

Routine	Fails when
sgcHTMLAssertContains(aHTML, aText)	aText does not appear verbatim in aHTML, or aText is empty (there is nothing to look for).
sgcHTMLAssertNotContains(aHTML, aText)	aText appears verbatim in aHTML, or aText is empty.
sgcHTMLAssertSelector(aHTML, aSelector, aExpectedCount = -1)	aSelector does not match exactly aExpectedCount elements, or matches none when aExpectedCount is -1.
sgcHTMLAssertAttr(aHTML, aSelector, aAttr, aExpected)	No element matches, the first match has no aAttr attribute, or its value, compared as written, is not aExpected.
sgcHTMLAssertText(aHTML, aSelector, aExpected)	No element matches, or the text of the first match is not aExpected. Runs of white space are collapsed to one space and trimmed on both sides first.
sgcHTMLAssertNoDoubleEscape(aHTML)	The markup contains `<`, `>`, `"`, `&` or `&#` (in any case): text that was HTML-encoded twice.
sgcHTMLAssertNoRawScript(aHTML, aUserText)	aUserText appears verbatim inside a script element, inside an on* event handler attribute or inside an htmx inline handler (an attribute whose name starts with hx-on, such as hx-on:click or hx-on::after-request). An empty aUserText always passes.

The messages quote at most 200 characters of the markup.

Snapshots

`sgcHTMLAssertSnapshot(aName, aHTML)` compares the output with a file saved by an earlier run, and `sgcHTMLSnapshotMatch(aName, aHTML)` does the same and returns a Boolean instead of raising.

- The first run writes `<sgcHTMLSnapshotFolder>\<aName>.snap.html` and passes. Keep the file with your tests, under version control.
- Later runs compare the output with that file. Both are normalized first: CR LF and CR become LF, trailing white space is removed from every line and trailing blank lines are dropped.
- On a mismatch the new output is written next to the snapshot as `<aName>.actual.html`, and the message gives the first differing line number, both lines and both file names. The .actual.html file is deleted once the output matches again.
- Run the tests with the environment variable `SGC_UPDATE_SNAPSHOTS=1` to accept the new output: every checked snapshot is overwritten and the checks pass.

- `sgcHTMLSnapshotFolder` is a global variable, by default a `snapshots` folder next to the executable, created when missing. aName may contain / to group snapshots in sub folders, and must not be empty. The files are UTF-8 without BOM, with LF line ends.

```
set SGC_UPDATE_SNAPSHOTS=1
MyProjectTests.exe
set SGC_UPDATE_SNAPSHOTS=
```

A DUnit Example

Testing Routes and Authentication without a Server

TsgcHTMX_Engine_Server.DispatchDirect needs no socket, so a test can run the whole request pipeline, sessions, route gates and CSRF included, and check the result with the assertions above.

To test a state-changing route, set the method and send the token of the session in the CSRF header: `oRequest.Headers.Values[FAuth.CSRFHeaderName] := oSession.CSRFToken`. Without it the engine answers 403.

See Also

- Routing
- Sessions, CSRF and Authentication
- TsgcHTMX_Engine_Server
- sgcHTML Nodes

Navigation and Live Updates

Boosted navigation, morph swaps, history, loading and error feedback, and the reconnect dialog, all from the page template and without writing JavaScript.

Introduction

TsgcHTMLTemplate_Bootstrap brings eight navigation options that turn a page rendered on the server into a single page application, with no JavaScript of your own. Every option is off by default and, while all of them are off, the page is rendered exactly as before. TsgcHTMLTemplate_HTMX inherits them, so they work with the htmx engine as well.

With any option on, the head of the page loads `/htmx.min.js`, the extensions that option needs and `/sgcHTMX.min.js`, all from the root of the site. No WebSocket is opened for them. TsgcHTMX_Engine_Server serves those files; another host can serve them with `sgcHTMXGetAsset`, see below.

Quick Start

Three lines give a site boosted navigation with a progress bar and an in-place morph swap:

```
oTemplate->Boost = true;
oTemplate->LoadingIndicator = true;
oTemplate->DefaultSwap = "morph:innerHTML";
```

With TsgcHTMX_Engine_Server, set them on its Template, or from the OnPrepareTemplate event of TsgcHTMLComponent_Site when the Site component builds the page.

The Options

Property	What the page gets
Boost	`hx-boost="true"` on the body and the head-support extension: links and forms are loaded with htmx and swapped into the body, and the head of a full page answer is merged. The Bootstrap script moves to the head.
PushURL	`hx-push-url="true"` on the body, inherited by every htmx request of the page.
DefaultSwap	`htmx.config.defaultSwapStyle`. A value that starts with `morph` also loads the idiomorph extension.
ViewTransitions	`htmx.config.globalViewTransitions` and the CSS at-rule `@view-transition{navigation:auto}`.

LoadingIndicator	A thin progress bar with the id `sgc-loading` at the top, and `hx-indicator="#sgc-loading"` on the body.
DisableOnSubmit	`data-sgc-disable-submit="true"` : the submit buttons of a form are disabled while its request runs.
ScrollRestoration	`data-sgc-scroll-restoration="true"` : the scroll position of every page is remembered and restored.
ErrorToast	A Bootstrap toast with the id `sgc-error-toast`, shown when a request fails.

Each property page describes what it emits in detail.

The Boosted Page Answer

A boosted link or form sends `HX-Request: true` and `HX-Boosted: true`. When such a request reaches the page of TsgcHTMX_Engine_Server, the engine answers with the encoded `<title>` of the template followed by the inner HTML of its body (GetBoostedPageHTML), and adds `Vary: HX-Request, HX-Boosted` to the response, so a cache keeps the boosted answer and the full document apart. Any other request for the page still gets the whole document.

In your own routes, `TsgcHTMXRequest.IsHTMX` and `TsgcHTMXRequest.IsBoosted` tell the two cases apart, so one route can return a fragment for an htmx request and a whole page for a plain one.

Response Headers from a Route

A route adds headers to its HTTP answer through `TsgcHTMXRequest.ResponseHeaders`, one `name=value` line each, which is how the htmx response headers are sent: `HX-Push-Url`, `HX-Redirect`, `HX-Location`, `HX-Trigger` or `HX-Retarget`.

Header names must be HTTP tokens, CR, LF and the other control characters are removed from the values, and the framing headers (Content-Length, Content-Type, Transfer-Encoding, Connection, Keep-Alive, Upgrade, Trailer and TE) are ignored. A route that returns no content but adds a valid header is still answered, with status 200 and an empty body. The list is ignored for WebSocket messages.

Morph Swaps and data-sgc-preserve

A morph swap updates the existing elements instead of replacing them, so the scroll position, the focus and the value being typed survive an update. It is used by **DefaultSwap** (a value that starts with `morph`), by `hx-swap="morph:innerHTML"` on a single element, and by the `hxMorph` swap mode of TsgcHTMX_Fragment, which emits `hx-swap-oob="morph"`.

sgcHTMX.min.js configures idiomorph so that the field with the focus keeps the value being typed, and an element that carries the `data-sgc-preserve` attribute is left untouched while it, or one of its descendants, has the focus. Put that attribute on an editor, a filter box or any region the user may be working in while the server pushes an update.

```
<div id="notes" data-sgc-preserve>
  <textarea name="notes"></textarea>
```

```
</div>
```

The Reconnect Dialog

TsgcHTMLTemplate_HTMX adds **ReconnectUI**, a Bootstrap modal with the id `sgc-reconnect`. sgcHTMX.min.js shows it when the realtime connection of the page closes while the page is not being left, and hides it when the connection is back, then dispatches `sgc:reconnected` on the body, so a region with `hx-trigger="sgc:reconnected from:body"` reloads itself with fresh data.

- After a dropped connection or a server restart the page reconnects by itself: the htmx WebSocket extension retries after the close codes 1006, 1012 and 1013 with its own backoff, and an event stream is retried by the browser.
- After an explicit server side disconnect (the server closed the connection with a normal close code) nothing retries, so the dialog stays open with its Reload button, which reloads the page.
- The texts come from the active locale: ReconnectTitle, ReconnectText and ReloadText.

Serving the Scripts on Another Host

`sgcHTMXGetAsset(aDocument, out aContentType, aContent)` returns the bundled assets by name: `htmx.min.js`, `htmx-ext-ws.min.js`, `htmx-ext-sse.min.js`, `idiomorph-ext.min.js`, `htmx-ext-head-support.min.js`, `sgcHTMX.min.js`, `bootstrap.min.css`, `bootstrap.bundle.min.js` and `chart.umd.min.js`. It returns False for any other path. The content holds the bytes of the file decoded with the default ANSI code page, so write it back with the same code page.

Preparing the Template from the Site Component

TsgcHTMLComponent_Site builds its own page template. Handle its **OnPrepareTemplate** event to set the navigation options, or anything else of the template, right before the page is rendered.

Texts

The navigation elements take their texts from the active locale (Localization): **LoadingText** for the label of the progress bar, **RequestFailedText** and **CloseText** for the error toast, and **ReconnectTitle**, **ReconnectText** and **ReloadText** for the reconnect dialog. Set them on `sgcHTMLDefaultLocale` to change the wording.

Notes and Limits

- Boost swaps the body, so the scripts of the body run again after every navigation. Keep the scripts that must run once in the head, or mark them with `hx-head="re-eval"` when they must run on every merge.
- htmx 2.0.4 does not honour a submit cancelled by an inline `onsubmit` handler: give such a form `hx-boost="false"`, or use `hx-confirm`.
- PushURL is inherited by every htmx request, so set `hx-push-url="false"` on the requests that return fragments.
- A 4xx answer that carries HTML is swapped like a page (a validation or a login page), while a 5xx answer is left to htmx, which swaps nothing and raises htmx:responseError.
- The navigation options need no WebSocket. They work on any host, the WebBroker engine included.

See Also

- TsgcHTMLTemplate_Bootstrap
- TsgcHTMLTemplate_HTMX
- WebSocket or SSE
- TsgcHTMX_Engine_Server
- Routing

WebSocket or SSE

How a page chooses between a WebSocket and a server-sent events stream, what the engine serves, and how messages, resumes and proxies behave.

Introduction

A page built with TsgcHTMX_Engine_Server receives its live updates over a WebSocket by default. The **Transport** property of TsgcHTMLTemplate_HTMX changes that to server-sent events (SSE), a plain HTTP response that stays open, or to an automatic choice. The markup of the page, the routes and the fragments stay the same: only the channel changes.

Transport	What the page does
htWebSocket (default)	The body carries `hx-ext="ws"` and `ws-connect="<WSEndpoint>"`, and the page loads the htmx WebSocket extension.
htSSE	The body carries `hx-ext="sse"`, `sse-connect="<SSEEndpoint>"` and `data-sgc-sse-post="<SSEPostEndpoint>"`, plus a hidden sink element that receives the events. No WebSocket is opened.
htAuto	The WebSocket markup plus the SSE attributes. sgcHTMX.min.js opens the WebSocket and switches to the event stream when it cannot be opened (after `sgcHTMX.config.sseFallbackAttempts` failures, 3 by default, or when the browser has no WebSocket support), and never switches back.

htAuto without a WSEndpoint renders htSSE, and without an SSEEndpoint htWebSocket. Use SSE when a proxy, a firewall or a hosting plan does not carry WebSocket traffic, and the WebSocket when the page sends a lot of data upstream.

Quick Start

1. Turn the transport on: set **SSE.Enabled** on the engine, or a **Transport** other than htWebSocket on its template, which enables the SSE endpoints as well.

2. Leave the endpoints at their defaults, or change them on the engine: the engine copies them into the template before every page render.

3. Push fragments exactly as with the WebSocket, with PushFragment and BroadcastFragment.

```
oHTMX->SSE->Enabled = true;
oHTMX->Template->Transport = htAuto;
oHTMX->Template->ReconnectUI = true;

oHTMX->BroadcastFragment("<div id=\"kpi\" hx-swap-oob=\"true\">" +
  CurrToStr(FRevenue) + "</div>");
```

The SSE Options of the Engine

What it does: the SSE property of the engine holds the whole transport configuration.

Property	Default	Description
Enabled	False	Accept and track the event streams opened on Endpoint. A Template.Transport other than htWebSocket turns the same machinery on without changing this property.
Endpoint	'/htmx-sse'	Path of the GET that opens the stream.
PostEndpoint	'/htmx-sse/send'	Path of the POST that carries the messages of the page.
EventName	'sgc'	Name of every pushed event. Empty sends unnamed events.
KeepAlive	15	Seconds without a write before a keepalive comment is sent. 0 turns it off.
ReplaySize	50	Events kept per stream for a resume. 0 turns the replay off.
ReplayTTL	60	Seconds the events of a closed stream are kept for a resume.

How a Stream Is Opened

1. The page opens `GET <SSE.Endpoint>` with `Accept: text/event-stream`. The engine turns on `Server.FallBack.ServerSentEvents` of the server (the previous value is restored when the engine is unhooked), so the server keeps the response open as an SSE connection.
2. The server writes the `retry:` interval and the GUID of the connection as the first message, which is what the page uses to send messages back.
3. The engine binds the stream to the session of its cookies, with the same rule as a WebSocket handshake: only the session cookie is read, and a handshake whose Origin does not match the Host header stays anonymous.
4. While the stream is idle the engine writes `: keepalive` every **SSE.KeepAlive** seconds, so proxies do not close it.

Pushes and Broadcasts

PushFragment with the GUID of a stream writes the fragment as one event: an `event:` line with **SSE.EventName**, an `id:` line with `<guid>:<seq>` and one `data:` line per line of the fragment. **BroadcastFragment** reaches every stream of the engine once, honouring its aExclude argument. A broadcast with a channel skips the streams, because an event stream cannot subscribe to a channel.

The page applies an event exactly as it applies a WebSocket message: an element with `hx-swap-oob` is swapped out of band, and a top level element without it replaces the element that has the same id.

Messages from the Page

An event stream only carries data from the server to the browser, so everything the page sends (`sgcHTMX.send`, `ws-send` and `data-sgc-ws-send`) is sent as a POST to **SSE.PostEndpoint**, with the GUID of the stream in the `X-SGC-Connection` header (the `sgc_connection` form or query field also works) and the message as the body. The CSRF token travels in the usual header when the page has a session.

Answer	When
204	The message was accepted and run as a message of that stream. A non empty answer is pushed back to the stream as an event.
403	The session of the POST did not carry a valid CSRF token.
404	The GUID is unknown, or the stream belongs to another session (an anonymous stream only matches an anonymous POST).
413	The body is larger than 1 MB.

The POST is an ordinary request, so any host that gives the engine a request path can serve it, WebBroker included.

Resuming a Stream

A browser reopens a dropped event stream by itself and sends the `Last-Event-ID` header with the id of the last event it received; sgcHTMX.min.js also adds a `lastEventId` query parameter. When that id is `<oldguid>:<n>` and the new stream belongs to the same session, the engine first sends the events after `n` again, with the ids of the new stream, so a short disconnection loses nothing.

The last **SSE.ReplaySize** events of a stream are kept, and for **SSE.ReplayTTL** seconds after it closes. Set ReplaySize to 0 to turn the replay off.

Hosts

Host	SSE
TsgcWebSocketHTTPServer	Supported. The WebSocket and the event streams live on the same server.
TsgcHTTPServer	Supported, and no WebSocket server is needed: the engine only needs a server that can keep a response open.
http.sys (TsgcHTMX_Engine_Server_HTTPAPI)	Supported, with the same options.

WebBroker, ISAPI, CGI and Apache modules	Not supported: the host owns the request and closes the response, so no stream can stay open. The POST endpoint still works, so a page there can only poll.

Behind a Proxy

- An event stream is one long HTTP response, so a proxy must not buffer it. With nginx, use `proxy_buffering off;` for the location of the stream, or send the response header `X-Accel-Buffering: no`, which has the same effect per response and is handy when you do not own the nginx configuration (add it in the OnSSECustomizeHeaders event of the connection).
- Cloudflare does not buffer `text/event-stream` responses, but it does close idle connections, so keep the keepalive comments the engine sends (SSE.KeepAlive, 15 seconds by default) and leave compression off for the stream.
- When a stream drops anyway, the browser reconnects by itself and sends Last-Event-ID, and the engine replays the events the page missed.

Writing Events by Hand

A connection whose Transport is `trpSSE` writes events with two methods of TsgcWSConnection: **WriteSSEEvent**(aEvent, aData, aID) writes the `event:`, `id:` and `data:` lines of one event, and **WriteSSEComment**(aText) writes a comment line, which browsers ignore and proxies see as traffic. Both write the whole block under the lock of the connection, so an event never interleaves with another write.

See Also

- TsgcHTMX_Engine_Server
- TsgcHTMLTemplate_HTMX
- Navigation and Live Updates
- Sessions, CSRF and Authentication

AI Data Pack

Ask a grid, a pivot or a form in plain language, and apply only what a validator let through.

Introduction

The AI Data Pack is the natural language layer of the sgcHTML data components. A human types a question over a grid, a pivot or a form, and TsgcHTMLAIAssistant turns it into a **validated operation set** that the component applies to its own rows. The question never reaches the database and never becomes SQL.

A provider is optional. With none configured, and on every platform where the AI units are not compiled in, a rule-based parser answers, so a page works with no API key at all.

The pack is made of the assistant, the operation model (TsgcHTMLGridOps, TsgcHTMLPivotOps and TsgcHTMLAIColumns), the new component TsgcHTMLComponent_InlineAIPrompt and the members the grid, the data table, the pivot and the form gained.

Quick Start

1. Create a **TsgcHTMLAIAssistant**. Configure a provider with Provider, APIKey and Model, or leave it empty and let the rule-based parser answer.

2. Assign it to the **AIAssistant** property of the data component and set **AIQueryEnabled** to True.

3. Dispatch the actions the controls post, as described below, and push the re-rendered component back as an out of band fragment.

```
FAI = new TsgcHTMLAIAssistant(this);
FAI->AllowRowDataToProvider = false;

oGrid->AIAssistant = FAI;
oGrid->AIQueryEnabled = true;

oGrid->LoadAIQueryState(vState);
oGrid->ProcessAIQuery(vQuery);
oGrid->LoadFromDataSource();
oHTMX->PushFragment(vGuid, oGrid->HTML);
```

What the Model Receives

A grid or pivot question sends the **column names, the captions, the types and up to three sample values per column**. Nothing else. No row of the dataset travels, whatever the question was.

Row data leaves the process only through the calls that are about rows, and only when **AllowRowDataToProvider** is True, which it is not by default:

Call	Sends rows	When AllowRowDataToProvider is False

QueryToGridOps, QueryToPivotOps	No	Works normally.
SuggestColumnValues, and the column assistant of the grid	Yes	Returns False and sends nothing.
Summarize	Yes	Returns False and sends nothing.
SmartPaste	Only the text a human pasted	Works normally, the dataset is not involved.

The system message the assistant sends tells the model to answer with JSON only and to ignore any instruction found in the question, in a column name or in a sample value.

Validation Is the Boundary

Everything that comes back from a provider is treated as hostile text. The answer is parsed and then:

- every column name must match the whitelist, by exact name and without case. No expressions, no aliases, no wildcards.
- every operator must be one of the closed operator set, and every aggregate one of sum, count, avg, min and max.
- every value is coerced by the data type of its column. A value that does not coerce drops its whole operation.
- a highlight style must be one of primary, secondary, success, danger, warning, info, light and dark. Anything else becomes the default style, so the class can never be free CSS.
- limits are clamped to MaxLimit, and the number of operations is capped.
- control characters are stripped, lengths are capped, and the free text explanation loses its angle brackets as well.
- anything unknown, at any level, is discarded in silence.

An operation set where nothing at all survived is cleared completely, the explanation included. So a prompt injection answer produces an empty or partial operation set, never an exception, and **no model text ever reaches the page** unless it survived validation, and even then it is written through the node layer and escaped.

The same boundary is applied to the state a browser sends back: **LoadAIQueryState** revalidates the JSON against the live column whitelist, so a state that names a column which no longer exists loses that operation.

The Rule-Based Parser

With no provider configured the assistant answers with its own parser, and it is also what answers a failed provider call while **UseRuleBasedFallback** is True. It reads clauses of the form column operator value joined by and or by a comma, plus sort by, group by, top N, between A and B and highlight ... in style, and for a pivot the rows, the columns and the aggregate. It knows the usual English spellings of each operator (greater than, over, at least, contains, starts with, is not, and so on) and it is tolerant about decimal separators, so a machine set to a comma reads a value written with a point.

The parser needs nothing that is Windows only, so it works on every platform, while the provider path needs the AI units. A question it does not understand produces no operation at all, and the page keeps the rows it had.

Caching

A provider answer is cached for **CacheMinutes** minutes, five by default. The key is the kind of call, the resolved language, the fingerprint of the column whitelist and the question, so the same question over a changed column set is a different entry. What is cached is the validated operation set, and it is validated again when it is read back. **ClearCache** empties it, and CacheMinutes 0 switches it off. Rule-based answers are not cached.

The Host Contract

The pack adds no HTTP route. Every control the components render is a `data-sgc-ws-send` form: the browser serializes it with window.sgcHTMX.send and sends it as a flat JSON object over the channel the page already has. The host dispatches on the `action` field, calls the matching method of the component named by the `table`, `form` or `prompt` field and answers with the re-rendered component as an out of band fragment.

Action	Fields	What the host calls
gridAIQuery	table, query, state	LoadAIQueryState(state) and ProcessAIQuery(query)
gridAIClear	table	ClearAIQuery
gridAIColumn	table, column	ProcessAIColumnValues(column)
gridAIAccept	table, column, row	AcceptAIColumnValue(row)
gridAIReject	table, column, row	RejectAIColumnValue(row)
pivotAIQuery	table, query, state	LoadAIQueryState(state) and ProcessAIQuery(query)
pivotAIClear	table	ClearAIQuery
formSmartPaste	form, text	ProcessSmartPaste(text)
inlineAIOpen	prompt, target	Open
inlineAIAsk	prompt, target, text	ProcessPrompt(text)
inlineAIAccept	prompt, target	Accept
inlineAIDiscard	prompt, target	Discard

The **table** field carries the TableID of the component, with one exception: the bar of a TsgcHTMLComponent_DataTable belongs to the grid it wraps, which is named `<SafeID(TableID)>_grid`, so compare the id instead of assuming it. The **form** field carries the FormID and the **prompt** field the PromptID.

The **state** field is only present when there is nowhere else to keep the query state: with *AIPersistState* True and a request session (see Sessions, CSRF and Authentication), the state lives in the session and nothing about it is written into the page. Without a session it travels in a hidden input of the bar, and it is revalidated every time it comes back.

Messages are not CSRF-checked, exactly as any other htmx WebSocket message, so keep the authorization of what an action does on the server, as always.

In the Grid

AIQueryEnabled renders the query bar, the interpreted as chip and the clear control. **ProcessAIQuery** answers a question, **OnAIQuery** still wins when the host handles it itself, and the validated set is kept in **QueryState** and applied every time the rows are produced. Filters, sorts, the grouping and the limit are applied to the rows the grid holds, and the highlights become contextual row and cell classes.

A column with an **AIMode** other than acOff renders a suggest control in its header. **ProcessAIColumnValues** asks the assistant for one value per visible row in a single call, and the answers become **AIProposals**, shown next to the current value with an Accept and a Reject control. The library writes nothing anywhere: an accepted proposal raises **OnAIColumnValues** and the host persists it.

That column call sends rows, so it refuses while AllowRowDataToProvider is False. A host that has no provider, or that produces the values itself, fills **AIProposals** directly, which is the supported way to show the same Accept and Reject half with no provider at all.

A grid bound to a dataset reloads with **LoadFromDataSource**, which reconciles the columns with the fields instead of throwing them away: a column whose Name still matches a field keeps everything the host configured on it, Width, Align, Title, AIMode and AIPrompt included, a column whose field is gone is removed and a new field is appended.

In the Pivot

The same bar, answered with QueryToPivotOps: the rows, the columns, the measure and the aggregate of the validated layout become the RowFields, the ColumnFields and the single Measure. The layout you configured at design time is **snapshotted before the first question** and restored when the query state is cleared, so ClearAIQuery brings your own pivot back exactly as it was.

In the Form

SmartPasteEnabled renders a paste zone above the form. **ProcessSmartPaste** hands the text to the assistant with the field whitelist (the name, the label and the type of every named field, and no value of the form), writes what came back into the fields by name and marks them as AI filled, which draws a small badge on the label and the valid styling on the control. **LoadValuesFromJSON** does the same from JSON you produced yourself, and **ClearAIFilled** drops the marks.

The pasted text keeps its line breaks on the way to the server, so an address or an invoice pasted as several lines arrives whole.

Localization

Nine texts of the pack come from Localization: **AIInterpretedAs**, **AISuggest**, **AIAccept**, **AIReject**, **AIDiscard**, **AIAsk**, **AIFilled**, **AIPasteHint** and **AIPasteButton**. Set them on sgcHTMLDefaultLocale, or on the locale of the request, as any other text.

Limits

- The components keep the question, the operation set and the pending proposals between two messages, so one set of components is one user. Key the components, or at least the query state, per session in a multi user application.
- SuggestColumnValues and Summarize need a provider and AllowRowDataToProvider True. They have no rule-based fallback.
- The provider path needs the AI units, which are Windows only. Everything else, the rule-based parser included, works on every platform the components work on.

- Nothing is ever written into your dataset by the library. Every accepted value reaches you through an event.

See Also

- TsgcHTMLAIAssistant
- TsgcHTMLComponent_InlineAIPrompt
- TsgcHTMLGridOps
- TsgcHTMLPivotOps
- TsgcHTMLAIColumns
- TsgcHTMLComponent_Grid
- TsgcHTMLComponent_PivotTable
- TsgcHTMLComponent_Form

Web Push

Reach a user whose browser is not on the page, with the application deciding who gets what.

Introduction

Web Push delivers a notification to a browser that is not showing the page, and even to one that is closed. The page subscribes once, the push service of the browser vendor keeps the subscription, and the application POSTs an encrypted notification to that subscription whenever it has something to say.

Editions. Web Push is available in the **Enterprise** and **All-Access** editions. The unit is compiled in only when SGC_WEBPUSH is defined, which sgcVer.inc sets for those two editions. The notification inbox and the preferences table are part of sgcHTML itself and work on every edition over WebSocket or SSE: only the push channel, which reaches a browser that is not on the page, needs the define.

Four pieces make it up, and each one can be used on its own:

Piece	What it does
TsgcHTMLWebPush	Validates what a browser posts when it subscribes, hands every subscription to the application, and sends notifications (RFC 8030, RFC 8291, RFC 8292).
TsgcHTMLComponent_NotificationInbox	The bell with the unread badge and the notifications of one user, in the page.
TsgcHTMLComponent_NotificationPreferences	The table a human ticks the channels in, one row per category.
TsgcHTMLComponent_Notification	Fans one notification out: in the page when the user is connected, as a push when the user is not.

Quick Start

1. Generate a VAPID pair once and store it. The public half is what the browser subscribes with, the private half signs the token the push service checks.

2. Create a TsgcHTMLWebPush, assign the pair and a Subject, and handle the three storage events. The component stores nothing itself.

3. Assign it to the WebPush property of TsgcHTMX_Engine_Server and set Template.WebPushEnabled. The engine then serves the subscribe and unsubscribe endpoints and the service worker, and the page carries the subscribe script.

4. Put a control carrying `data-sgc-webpush` in the page, so the permission prompt runs from a user gesture, which is the only way a browser accepts it.

```
TsgcHTMLWebPush::GenerateVAPIDKeys(vPublic, vPrivate);
```

```
FPush = new TsgcHTMLWebPush(this);
FPush->VAPIDPublicKey = vPublic;
FPush->VAPIDPrivateKey = vPrivate;
FPush->Subject = "mailto:support@example.com";
FPush->Enabled = true;

oHTMX->WebPush = FPush;
oHTMX->Template->WebPushEnabled = true;

FPush->Send("u1", "Order shipped", "Order 1042 left the warehouse", "/orders/1042");
```

The page side is one attribute:

```
<button data-sgc-webpush>Enable notifications</button>
<button data-sgc-webpush="unsubscribe">Turn them off</button>
```

window.sgcWebPush.subscribe and window.sgcWebPush.unsubscribe are also callable from your own script, and both answer a promise.

What Travels, and Where It Is Stored

A subscription is three strings: the endpoint URL of the push service, the public key of the browser (p256dh) and its auth secret. The application stores them, keyed by the user, and hands them back when it is time to send. **sgcHTML stores none of them**, so nothing is lost when the process restarts and nothing needs to be shared between the servers of a farm.

The notification itself is encrypted on the server with a key pair generated for that one message (RFC 8291), so the push service forwards bytes it cannot read. The VAPID token (RFC 8292) is what identifies the application server to the service, and it is signed with the private half of the pair.

Who Is Asking

The user a subscription belongs to is NEVER read from what the browser posted. The engine resolves it in one of two ways:

- with a TsgcHTMLAuth assigned, from the session of the session cookie, and the CSRF rule of the routes applies to the POST as well. A remember-me cookie is never redeemed here.
- without one, through OnResolveUser, so an application that runs its own sessions answers with its own user id and can refuse the request.

With neither of them the request is refused with 403. A subscription with no user behind it is worse than a refusal, because nothing can ever be sent to it and nothing can ever delete it.

The Endpoints the Engine Serves

Request	Answer
POST /push/subscribe	204 when the subscription was handed to the host, 400 when the body is not a valid subscription, 403 when the request resolves to no user, 413 when the body is larger than 64 KB.

POST /push/unsubscribe	The same, with the endpoint alone.
GET the service worker path	The service worker, which carries the push and notificationclick handlers once WebPushEnabled is on. It is served whether or not the page is a PWA.

Both paths are configurable, and the engine copies them into the template before every render, so the page and the server can never drift apart. An application that routes the two POSTs itself calls HandleSubscribe and HandleUnsubscribe instead, and clears the two endpoint properties.

The service worker path is Template.ServiceWorkerPath, `/sw.js` by default. An application mounted under a prefix sets its own, and both the registration in the page and the path the engine answers on follow it.

In the Page, or as a Push

TsgcHTMLComponent_Notification decides between the two. Given a user, it records the notification exactly as it always did, and then:

- when the user is connected, LastFragmentHTML is left holding the out of band markup of the new notification plus the refreshed badge, which is what the host pushes to that user's page.
- when the user is NOT connected, a WebPush is assigned and the host answered OnGetChannel with the push channel, one notification is sent. Otherwise nothing leaves the page.

Presence is what decides whether a user is connected, and with no Presence assigned every user counts as connected, which is what the component did before. OnGetChannel arrives as the in app channel only, so **the library never guesses an opt in**: an application that does not handle the event never sends a push, whatever it stores in its preferences table.

What a Browser Needs

- a secure context: https, or localhost while developing.
- a service worker, which the engine serves and the page registers.
- a permission the human granted from a gesture. A prompt shown on load is refused by the browser itself.
- userVisibleOnly: every push must show a notification, which is what the bundled worker does.

A push service answers 404 or 410 for a subscription that is gone for good, because the human revoked the permission, cleared the site data or uninstalled the application. That raises OnSubscriptionExpired, and the row should be deleted: keeping it costs a failed request per notification.

ASP.NET Core Razor (Tag Helpers)

Use the sgcHTML components inside ASP.NET Core MVC and Razor Pages views with Tag Helpers or the fluent Html.Sgc() helpers, with the data served by ordinary controller actions or Minimal API endpoints.

Introduction

Applies to sgcHTML .NET. The Razor layer ships in the **esegece.sgcHTML.AspNetCore** NuGet package (and its **esegece.sgcHTML.AspNetCore.Community** twin, which includes it too) for net8.0 and net9.0, namespace `esegece.sgcWebSockets.AspNetCore.Razor`. Delphi and C++ Builder have no Razor, so there is no Delphi counterpart.

The Razor layer lets an ASP.NET Core MVC or Razor Pages application use the sgcHTML components inside its own views, the way Telerik UI for ASP.NET Core and DevExpress ASP.NET Core are used. You write `<sgc-grid>`, `<sgc-textbox asp-for="...">` or `<sgc-chart>` in a `.cshtml` file, or the fluent twin `@Html.Sgc().Grid<T>()`, and serve the data from an ordinary controller action or Minimal API endpoint.

The difference is under the hood. There is no JavaScript widget framework and no JSON data source: every component renders server side HTML, and htmx swaps the fragments your actions return. Live charts are updated over the sgcHTML WebSocket, with no SignalR code.

Mental Model for Telerik and DevExpress Users

- **Markup in your views.** Components are Tag Helpers (`sgc-` prefix) or fluent helpers under `Html.Sgc()`. Both fill the same definition, so they render the same HTML.
- **One data action per grid.** The grid sends every page, sort, filter and search click to one action that receives an `SgcDataRequest` (the `DataSourceRequest` equivalent) and answers `this.SgcGrid(request, query)` over an `IQueryable<T>` (the `ToDataSourceResult` equivalent). EF Core translates the query to SQL.
- **Model binding with asp-for.** Inputs read the name, value, label, `required`, `maxlength` and the `ModelState` errors from your model, like the built-in Tag Helpers.
- **Stateless.** The grid columns written in the view travel with each request inside a small protected token, so the data action needs no second column definition, and it works on web farms and after an app restart.

Setup

Install the package with `dotnet add package esegece.sgcHTML.AspNetCore` (or `esegece.sgcHTML.AspNetCore.Community`), then register the services and the middleware in **Program.cs**. `AddSgcHtml` with `ServeRootPage = false` lets your MVC app own `/` while sgcHTML still serves its assets and the `/ws` push channel. `AddSgcHtmlRazor` registers the `SgcDataRequest` model binder, the grid state protector (ASP.NET Core Data Protection) and the `/sgcHTML.css` bundle.

Make the Tag Helpers and the fluent helpers visible to every view in **Views/_ViewImports.cshtml**:

Add the styles and the scripts to **Views/Shared/_Layout.cshtml**. `<sgc-styles />` writes Bootstrap 5 and `/sgcHTML.css`, the CSS of every sgc component, built once at startup and served with an ETag. `<sgc-scripts />` writes htmx, Bootstrap JS and the sgcHTMX runtime.

If the application already has its own Bootstrap build or theme, turn the bundled copy off and keep only the sgc CSS:

Tag	At-tribute	Description
<sgc-styles>	boot-strap	Writes the embedded Bootstrap 5 CSS. Default `true`. `/sgcHTML.css` is al-ways written.
<sgc-scripts>	live	The live transport: `false` (default, none), `ws` (htmx WebSocket extension, `true` is accepted as `ws`) or `sse` (htmx SSE extension). Needed only by live charts.
<sgc-scripts>	boot-strap	Writes the embedded Bootstrap JS bundle. Default `true`.

Components

Tag Helper	Fluent helper	Topic
<sgc-grid>, <sgc-column>	Html.Sgc().Grid<T>()	Razor Grid: server paging, sort, filters, search, export, and the data contract.
<sgc-form>, <sgc-textbox>, <sgc-textarea>, <sgc-select>, <sgc-check-box>, <sgc-datepicker>, <sgc-submit>	TextBoxFor, TextAreaFor, Se-lectFor, CheckBoxFor, DatePickerFor	Razor Forms and Bound In-puts
<sgc-autocomplete>	AutoCompleteFor	Razor AutoComplete
<sgc-dialog>, sgc-dialog="name"	Html.Sgc().Dialog()	Razor Dialog
<sgc-chart>, <sgc-series>	Html.Sgc().Chart()	Razor Chart: static, polling and live.

Every action helper has a Minimal API twin on the static class `SgcResults`, and `SgcDataRequest` binds in a Minimal API endpoint through its `BindAsync` method:

MVC controller	Minimal API	Answers
this.SgcGrid(request, query)	SgcResults.Grid(request, query)	The grid fragment (rows, pager, header).
this.SgcGridExport(request, query, ...)	SgcResults.GridExport(request, query, ...)	An XLSX or PDF file of the fil-tered, sorted rows.
this.SgcAutoComplete(items)	SgcResults.AutoComplete(items)	The suggestions list.
this.SgcCloseDialog(name, re-fresh)	SgcResults.CloseDialog(name, re-fresh)	Closes a dialog and optionally reloads a grid.

this.SgcRedirect(url)	SgcResults.Redirect(url)	An `HX-Redirect` : htmx navigates the whole page.
this.SgcChart(builder)	SgcResults.Chart(builder)	The data of a polling chart.

Coming from Telerik or DevExpress

Telerik UI for ASP.NET Core / DevExpress	sgcHTML Razor
<kendo-grid> / Html.Kendo().Grid<T>() / Html.DevExpress().GridView()	<sgc-grid> / Html.Sgc().Grid<T>()
<column field="..." /> / columns.Bound(o => o.X)	<sgc-column field="..." /> / c.Bound(o => o.X)
[DataSourceRequest] DataSourceRequest request	SgcDataRequest request (bound by AddSgcHtmlRazor)
Json(query.ToDataSourceResult(request))	this.SgcGrid(request, query) (HTML fragment), or query.ToSgcDataResult(request) for the data only
Grid Excel / PDF export (.Excel(), .Pdf())	export="xlsx,pdf" + this.SgcGridExport(request, query)
Html.Kendo().TextBoxFor(m => m.X) / <kendo-textbox for="X">	Html.Sgc().TextBoxFor(m => m.X) / <sgc-textbox asp-for="X">
DropDownListFor, DatePickerFor, CheckBoxFor	SelectFor, DatePickerFor, CheckBoxFor (<sgc-select>, <sgc-datepicker>, <sgc-checkbox>)
ComboBoxFor / AutoCompleteFor with a Read action	<sgc-autocomplete asp-for search-action> + this.SgcAutoComplete(items)
<kendo-window> / PopupControl + grid popup editing	<sgc-dialog> + sgc-dialog="name" + this.SgcCloseDialog(name, refresh)
<kendo-chart> / Html.DevExpress().Chart()	<sgc-chart> + <sgc-series> / Html.Sgc().Chart()
Chart refresh through a DataSource plus a timer, or SignalR	refresh-every="5s" (polling) or live="true" + hub.BroadcastChartAsync (push)

SQLite Notes

The grid, the filters and the search run as EF Core queries, so the database provider decides two details that surprise people on SQLite:

- **No ORDER BY on decimal.** EF Core SQLite cannot sort a `decimal` column, so a grid sort on a decimal property fails. Store money as `double` on SQLite, or keep `decimal` on SQL Server, PostgreSQL or MySQL, which sort it fine.
- **Contains is case sensitive.** EF Core translates `string.Contains` to SQLite `instr()`, which matches case, so the grid text filters and the search box find `Acme` but not `acme`. For a lookup that should ignore case, query with `EF.Functions.Like`, which SQLite compares case insensitively for ASCII. Other databases follow the column collation (for example, case insensitive on the SQL Server default collation).

Demo

The **demos\61.HTML.AspNetCore\22.RazorMvc** demo is a complete MVC application on EF Core SQLite: an orders grid with server paging, filters, search, export and `push-url`, editing in a dialog with an autocomplete customer lookup, a create form written with the fluent helpers, a client side customer grid, and a dashboard with static, polling and live charts. Its README carries the same Telerik and DevExpress mapping table.

See Also

- Razor Grid
- Razor Forms and Bound Inputs
- Razor AutoComplete
- Razor Dialog
- Razor Chart
- HTML Components
- WebSocket or SSE

Razor Grid

The sgc-grid Tag Helper and Html.Sgc().Grid<T>(): server paging, sorting, filters, search and export over an IQueryable, with the SgcDataRequest data contract.

Introduction

Applies to sgcHTML .NET. The Razor layer ships in the **esegece.sgcHTML.AspNetCore** NuGet package (and its **esegece.sgcHTML.AspNetCore.Community** twin, which includes it too) for net8.0 and net9.0, namespace `esegece.sgcWebSockets.AspNetCore.Razor`. Delphi and C++ Builder have no Razor, so there is no Delphi counterpart.

`<sgc-grid>` renders a Bootstrap table with server paging, sorting, column filters, a global search box, XLSX and PDF export, and optional address bar history. Every interaction is an htmx GET to one data action, which runs the query on the database and answers the new grid fragment. The fluent twin is `Html.Sgc().Grid<T>()`. Page 1 can be rendered inline from the view model, so the page shows data at once with no loading flash.

The examples use this row type, an `IQueryable` projection of an EF Core entity, so the sort, filters, search and paging of the projection are translated to SQL:

Tag Helper

Fluent Helper

Controller Actions

The view action passes the query to the view, which takes page 1 from it. Every later click of the grid lands on **Read**. The toolbar export links land on **Export**, with the current filters and sort and a `format=xlsx` or `format=pdf` parameter.

Minimal API

Use `read-url` and `export-url` (fluent `ReadUrl` and `ExportUrl`) instead of the action attributes. Without `items` the grid renders empty and loads page 1 from the endpoint as soon as the page opens.

Client Side Grid

A small, fixed list needs no endpoint. Hand it to `for` (a model expression) or to the fluent `BindTo`: every row is rendered at once and the headers sort in the browser. Without `<sgc-column>` children the public simple properties of the row type become the columns.

<sgc-grid> Attributes

Attribute	Fluent	Description
name	Grid<T>(name)	The grid name: the element id, and the name that `SgcCloseDialog(..., refresh)` reloads.
key	Key(o => o.Id)	The key field. It is the default order when no sort is requested, so paging stays deterministic.
asp-action, asp-controller, asp-area, asp-route-*	Read(action, controller, routeValues)	The data action, resolved like the anchor Tag Helper.
read-url	ReadUrl(url)	A raw data URL, for Minimal API endpoints.
items	Items(rows)	An `IQueryable<T>` or `IEnumerable<T>` of the first render: page 1 is taken from it. Null renders an empty grid that loads page 1 from the data endpoint.
for	BindTo(rows)	A fixed in memory list: every row rendered, sorted in the browser, no endpoint and no state.
page-size, page-sizes	Pageable(pageSize, pageSizes)	The default page size (20 when omitted) and the choices of the page size selector, comma separated. A requested page size is clamped to these choices.
sortable, filterable, searchable	Sortable(), Filterable(), Searchable()	Turn on header sorting, the filter row and the global search box.
push-url	PushUrl()	Adds `hx-push-url`, so the address bar keeps page, sort and filters, and reload, back and bookmarks work.
export	Export(formats)	The toolbar export links: `xlsx`, `pdf`, comma separated.
asp-export-action, asp-export-controller	ExportAction(action, controller)	The export action. The controller defaults to `asp-controller`.
export-url	ExportUrl(url)	A raw export URL, for Minimal API endpoints.

<sgc-column> Attributes

Attribute	Fluent	Description
field	Bound(o => o.X) or Add("X")	The property of the row.
title	Title(text)	The header text. Default: the field.
format	Format(format)	A composite format such as `{0:N2}`, `{0:d}` or `{0:yyyy-MM-dd}`.
width	Width(pixels) or Width(css)	The column width. A plain number is pixels.
align	Align(value)	`left`, `center` or `right`.
sortable, filterable, searchable	Sortable(bool), Filterable(bool), Searchable(bool)	Per column switches. Sortable and filterable default to the grid setting. Searchable marks the columns the search box looks in.
filter	Filter(SgcGridFilterKind)	`text` (default, one input) or `range` (from and to inputs, for dates and numbers).
encode	Encode(bool)	Whether the cell text is HTML encoded. Default `true`. `false` writes the value as markup, for example a button built by a computed property.

Data Contract

`SgcDataRequest` is what the grid sends. MVC binds it through the model binder that `AddSgcHtmlRazor` registers, and Minimal API binds it through its static `BindAsync(HttpContext)`. Malformed input never throws: invalid numbers fall back to the defaults.

Query parameter	Example	Meaning
page	page=3	The page, 1 based.
pageSize	pageSize=25	The page size, clamped to the page sizes the grid allows.
sort	sort=Customer,-OrderDate	A comma list of fields. A `-` prefix means descending.
f.<Field>	f.Customer=acme	A column filter. Strings use contains. Numbers, booleans and enums use equals.
f.<Field>.from, f.<Field>.to	f.OrderDate.from=2026-01-01	A range filter for dates and numbers.

q	q=madrid	The global search: an OR across the columns marked searchable.
sgc	(opaque)	The protected grid state, see below.

Member	Type	Description
SgcDataRequest.Page	int	The requested page, 1 based.
SgcDataRequest.PageSize	int	The requested page size.
SgcDataRequest.Sort	IReadOnlyList<SgcSort>	The sort fields, each with `Field` and `Descending`.
SgcDataRequest.Filters	IReadOnlyList<SgcFilter>	The column filters, each with `Field`, `Operator`, `Value`, `From` and `To`.
SgcDataRequest.Search	string	The global search text.
SgcDataRequest.State	SgcGridState	The unprotected grid definition, or null when the request carries none.
SgcDataResult<T>.Data	IReadOnlyList<T>	The rows of the page.
SgcDataResult<T>.Total	int	The row count after filters and search, before paging.
SgcDataResult<T>.Page, PageSize	int	The page and page size that were applied.

`ToSgcDataResult` is the data only step, the equivalent of `ToDataSourceResult`. Use it to feed your own renderer or a JSON client:

Protected Grid State

The columns, formats, page sizes and the fields that may be sorted and filtered are written in the view, yet the data action must render the same grid. The Tag Helper serialises that definition into a small token, protects it with ASP.NET Core Data Protection (purpose `sgcHTML.Razor.Grid`) and sends it back on every request as the `sgc` parameter. There is no server side registry, so it survives an app restart. On a web farm, share the Data Protection key ring between the servers, as you already do for antiforgery tokens. Sorting and filtering apply only to the fields listed in the protected state, so a tampered URL cannot reach other columns. `SgcGrid` and `SgcResults.Grid` answer 400 when the state is missing or invalid.

Query Building

`ToSgcDataResult` builds expression trees, never dynamic LINQ strings, so EF Core translates them to parameterised SQL. The order is filter, search, count, sort, then skip and take. Values are converted with the invariant culture. Unknown fields are ignored rather than raising, so a stale bookmark degrades gracefully. When no sort is requested and the source is not already ordered, the `key` column orders the rows.

Export

`this.SgcGridExport` and `SgcResults.GridExport` apply the current filters, search and sort without paging, and answer an XLSX or PDF file. Without an explicit `SgcExportFormat` the format comes from the `format` parameter of the toolbar link, and an unknown format answers 400. The optional `fileName` names the download, and `maxRows` caps the rows (default 10000).

SQLite Notes

- **No ORDER BY on decimal.** EF Core SQLite cannot sort a `decimal` column, so a grid sort on a decimal property fails. Store money as `double` on SQLite, or keep `decimal` on SQL Server, PostgreSQL or MySQL, which sort it fine.
- **Contains is case sensitive.** EF Core translates `string.Contains` to SQLite `instr()`, which matches case, so the grid text filters and the search box find `Acme` but not `acme`. For a lookup that should ignore case, query with `EF.Functions.Like`, which SQLite compares case insensitively for ASCII. Other databases follow the column collation (for example, case insensitive on the SQL Server default collation).

See Also

- ASP.NET Core Razor (Tag Helpers)
- Razor Forms and Bound Inputs
- Razor AutoComplete
- Razor Dialog
- Razor Chart
- TsgcHTMLComponent_Grid

Razor Forms and Bound Inputs

Model bound inputs with asp-for (sgc-textbox, sgc-textarea, sgc-select, sgc-checkbox, sgc-datepicker) and their fluent twins, plus sgc-form hx for htmx validation and SgcRedirect.

Introduction

Applies to sgcHTML .NET. The Razor layer ships in the **esegece.sgcHTML.AspNetCore** NuGet package (and its **esegece.sgcHTML.AspNetCore.Community** twin, which includes it too) for net8.0 and net9.0, namespace `esegece.sgcWebSockets.AspNetCore.Razor`. Delphi and C++ Builder have no Razor, so there is no Delphi counterpart.

The bound inputs are the sgcHTML twins of the ASP.NET Core input Tag Helpers. Each takes `asp-for`, a model expression, and reads from it the same data the built-in Tag Helpers use: the name, id and value, the label from `[Display]`, `required` from `[Required]`, `maxlength` from `[StringLength]`, and the error message from `ModelState`, shown under the input. `<sgc-form hx="true">` posts the form with htmx and swaps it in place, which is the canonical htmx validation flow.

The examples bind this model:

Tag Helpers

Fluent Helpers

The same fields with the fluent helpers. They need a typed view (`@model`), because `Html.Sgc()` on an `IHtmlHelper<TModel>` is what offers the `...For` methods.

Controller Actions

An invalid post returns the form partial, and htmx swaps it in place with the errors under each input. A valid post answers a redirect or a dialog close. The antiforgery token is written by `<sgc-form>` and sent by `hx-post`, so `[ValidateAntiForgeryToken]` works as usual.

`this.SgcRedirect(url)` answers an `HX-Redirect` header, so htmx navigates the whole page, like a 302 after a classic post. Inside a dialog, answer `this.SgcCloseDialog(name, refresh)` instead, see Razor Dialog.

Minimal API

`SgcResults.Redirect` is the Minimal API twin of `SgcRedirect`:

Reference

Tag Helper	Fluent	Attributes
<sgc-textbox asp-for>	TextBoxFor(m => m.X)	`placeholder` (`Placeholder`), `type` (`Type`). When omitted, the type is inferred from the model: `[DataType]` gives `email` , `password` , `tel` or `url` , and an integer property gives `number` . A `float` , `double` or `decimal` property renders `type="text" inputmode="decimal"` , because a number input without `step="any"` rejects values such as 99.90, while phones still show the decimal keyboard.
<sgc-textarea asp-for>	TextAreaFor(m => m.X)	`rows` (`Rows`), `placeholder` (`Placeholder`).
<sgc-select asp-for>	SelectFor(m => m.X, items)	`asp-items` , an `IEnumerable<SelectListItem>` . `placeholder` (`Placeholder`).
<sgc-checkbox asp-for>	CheckBoxFor(m => m.X)	Binds a `bool` .
<sgc-datepicker asp-for>	DatePickerFor(m => m.X)	`min` , `max` (yyyy-MM-dd, `Min` , `Max`), `mode` : `date` (default), `time` or `datetime` (`Mode`).
<sgc-auto-complete asp-for>	AutoCompleteFor(m => m.X)	See Razor AutoComplete.
<sgc-form>		`asp-action` , `asp-controller` , `asp-area` , `asp-route-*` : the post target. `method` : post by default. `hx` : `true` posts with htmx (`hx-post`) and swaps the form outerHTML. `target` : the htmx target, the form itself by default.
<sgc-submit>		`text` : the caption (the child content is used when empty). `css` : the CSS class, `btn btn-primary` by default.

The fluent helpers write the same markup as the Tag Helpers and can be mixed freely inside one `<sgc-form>`. They wrap the core **TsgcHTMLComponent_Edit**, **TsgcHTMLComponent_Memo**, **TsgcHTMLComponent_Select**, **TsgcHTMLComponent_CheckBox** and **TsgcHTMLComponent_DatePicker** components.

See Also

- ASP.NET Core Razor (Tag Helpers)
- Razor Grid
- Razor AutoComplete

- Razor Dialog
- Razor Chart

Razor AutoComplete

The sgc-autocomplete Tag Helper and AutoCompleteFor: a key bound lookup input fed by a search action that answers SgcAutoComplete.

Introduction

Applies to sgcHTML .NET. The Razor layer ships in the **esegece.sgcHTML.AspNetCore** NuGet package (and its **esegece.sgcHTML.AspNetCore.Community** twin, which includes it too) for net8.0 and net9.0, namespace `esegece.sgcWebSockets.AspNetCore.Razor`. Delphi and C++ Builder have no Razor, so there is no Delphi counterpart.

`<sgc-autocomplete>` is a lookup input bound to a key, like the Telerik ComboBox. The visible input carries the text the user types, a hidden input named after `asp-for` carries the key of the chosen item, so the form posts the key. While the user types, the component sends an htmx GET to the search action (after `min-length` characters and a `debounce` pause), and the action answers the suggestions with `this.SgcAutoComplete(items)`.

Tag Helper

Fluent Helper

`text` (`Text`) is the display text of the current key, so an edit form shows the customer name while it posts `CustomerId`. Look it up in the GET action and pass it through `ViewBag` or the view model.

Controller Action

The component sends the typed text as the `query` parameter. Answer a sequence of `SgcItem(Value, Text)`: `Value` is the key that is posted, `Text` is what the list and the input show.

Minimal API

Use `search-url` (fluent `SearchUrl`) instead of `search-action`, and answer `SgcResults.AutoComplete`:

The result is an out of band fragment for the suggestions panel of the component that asked. The component identifies itself with a parameter it sends automatically, so a request typed by hand in the browser, without that parameter, answers 400.

Attributes

Attribute	Fluent	Description
asp-for	AutoCompleteFor(m => m.X)	The key property. Its name is used by the hidden input, and the label and validation come from the model as for the other bound inputs.
search-action, search-controller	Search(action, controller)	The search action.
search-url	SearchUrl(url)	A raw search URL, for Minimal API endpoints.
min-length	MinLength(n)	The minimum text length before searching. Default 1.
debounce	Debounce(ms)	The pause in milliseconds after the last key stroke. Default 250.
max-results	MaxResults(n)	The maximum number of suggestions. Default 10. `SgcAutoComplete(items, maxResults)` can also cap it server side.
placeholder	Placeholder(text)	The placeholder of the visible input.
text	Text(text)	The initial display text of the current key.

SQLite Notes

The action above uses `EF.Functions.Like` rather than `Contains` on purpose: on SQLite `Contains` becomes a case sensitive `instr()`, while `LIKE` ignores case for ASCII, which is what a lookup should do. On SQL Server, PostgreSQL or MySQL the column collation decides.

See Also

- ASP.NET Core Razor (Tag Helpers)
- Razor Grid
- Razor Forms and Bound Inputs
- Razor Dialog
- Razor Chart
- TsgcHTMLComponent_AutoComplete

Razor Dialog

The sgc-dialog Tag Helper, the sgc-dialog trigger attribute and SgcCloseDialog: load a form into a modal, validate it in place, then close the dialog and reload a grid.

Introduction

Applies to sgcHTML .NET. The Razor layer ships in the **esegece.sgcHTML.AspNetCore** NuGet package (and its **esegece.sgcHTML.AspNetCore.Community** twin, which includes it too) for net8.0 and net9.0, namespace `esegece.sgcWebSockets.AspNetCore.Razor`. Delphi and C++ Builder have no Razor, so there is no Delphi counterpart.

`<sgc-dialog>` declares a Bootstrap modal once in the page. Any element with the `sgc-dialog="name"` attribute loads its URL into that dialog with htmx and opens it. Combined with <sgc-form hx="true"> and <sgc-grid>, this is the Telerik "edit the grid row in a window" pattern: the form posts inside the dialog, an invalid post shows its errors in place, and a valid post closes the dialog and reloads the grid.

Tag Helper

The `sgc-dialog` attribute works on any element and resolves `asp-action`, `asp-controller`, `asp-area` and `asp-route-*` like the anchor Tag Helper. It writes `hx-get` targeting the dialog body, and the dialog opens once the content has been swapped in.

Fluent Helper

Controller Actions

The GET action returns the form as a partial view, which lands in the dialog body. The POST action returns the same partial when the model is invalid, and `this.SgcCloseDialog(name, refresh)` when it is saved.

`SgcCloseDialog` answers 200 with an empty body, `HX-Reswap: none` (the form stays in place while the dialog closes) and `HX-Trigger: {"sgc-dialog-close":{"name":...,"refresh":...}}`. The dialog script closes the dialog and fires `sgc-refresh` on the `refresh` element. For a grid, pass its name, and the grid reloads its current page with the current sort and filters. Leave `refresh` null to only close the dialog.

Minimal API

Attributes

Attribute	Fluent	Description
name	Dialog(name)	The dialog name, used by `sgc-dialog="name"` and `SgcCloseDialog(name)`.
title	Title(text)	The dialog title.
size	Size(value)	`sm`, `lg`, `xl` or `fullscreen`.

The dialog wraps the core TsgcHTMLComponent_Modal.

See Also

- ASP.NET Core Razor (Tag Helpers)
- Razor Grid
- Razor Forms and Bound Inputs
- Razor AutoComplete
- Razor Chart
- TsgcHTMLComponent_Modal

Razor Chart

The sgc-chart Tag Helper and Html.Sgc().Chart(): static charts, polling charts with refresh-every, and live charts pushed over the sgcHTML WebSocket with BroadcastChartAsync.

Introduction

Applies to sgcHTML .NET. The Razor layer ships in the **esegece.sgcHTML.AspNetCore** NuGet package (and its **esegece.sgcHTML.AspNetCore.Community** twin, which includes it too) for net8.0 and net9.0, namespace `esegece.sgcWebSockets.AspNetCore.Razor`. Delphi and C++ Builder have no Razor, so there is no Delphi counterpart.

`<sgc-chart>` renders a Chart.js chart with one `<sgc-series>` per dataset. The fluent twin is `Html.Sgc().Chart(name)`. A chart can be static (the data is rendered with the page), polling (htmx asks an action again at an interval) or live (the server pushes new data over the sgcHTML WebSocket to every open browser).

The `name` is required. It is the canvas id, and it is the key that polling answers and live updates match on.

Static

`labels` takes an `IEnumerable<string>` and `data` an `IEnumerable<double>`. The view model here is a plain class with `string[] Months` and `double[] Sales`, `Orders` and `ByStatus`.

Polling

`refresh-action` (or `refresh-url`) plus `refresh-every` makes the chart ask the action again with `hx-get`. The interval takes `5s`, `500ms`, `1m` or a plain number of seconds, default `5s`. When the chart has series, the first data renders with the page and the polls start after the first interval. When it has no series, the chart asks the action as soon as the page loads, then at every interval. `Statuses` is a static field of the `DashboardController` shown below.

The action answers `this.SgcChart(builder)`. `SgcChartBuilder` is the same builder as the fluent helper, created directly:

Minimal API

Live

`live="true"` (fluent `Live()`) subscribes the chart to the sgcHTML WebSocket, so the layout must write `<sgc-scripts live="ws" />` and the pipeline must call `app.UseWebSockets()` before `app.UseSgcHtml()`, see Setup. Server code then updates every open chart with that name through `ISgcHtmlHub`. Telerik and DevExpress need SignalR plumbing for this; here it is one call.

A background service that pushes a new point once a second. Register it with `builder.Services.AddHostedService<LiveSalesFeed>()`:

`BroadcastChartAsync` replaces the labels and the series of every open chart named `live-orders`. An overload takes an `excludeConnectionId`. To update the chart of one browser only, use `PushChartAsync` with its connection id:

Attributes

Attribute	Fluent	Description
name	Chart(name)	Required. The canvas id and the key of polling answers and live updates.
type	Type(type)	line (default), bar, pie, doughnut, area and the other sgcHTML chart types.
height, width	Height(pixels or css), Width(pixels or css)	The CSS size. A plain number is pixels. The default height is 400.
labels	Labels(labels)	The category labels, an `IEnumerable<string>`.
stacked	Stacked()	Stacks the series.
options	Options(json)	Raw Chart.js configuration appended to the config. It is written as is, so pass trusted markup only.
refresh-action, refresh-controller	Refresh(action, controller)	The action a polling chart asks.
refresh-url	RefreshUrl(url)	A raw polling URL, for Minimal API endpoints.
refresh-every	RefreshEvery(value)	The polling interval: `5s`, `500ms`, `1m` or seconds. Default `5s`.
live	Live()	Listens to `BroadcastChartAsync` and `PushChartAsync` over the sgcHTML WebSocket.

<sgc-series> attribute	Fluent	Description
name	Series(name, ...)	The legend text.
data	Series(..., data, ...)	The values, an `IEnumerable<double>`.
color	Series(..., color)	A CSS color for the line and the fill.

type	Series(..., type)	A per series type for mixed charts: line, bar, area, scatter or bubble.

See Also

- ASP.NET Core Razor (Tag Helpers)
- Razor Grid
- Razor Forms and Bound Inputs
- Razor AutoComplete
- Razor Dialog
- TsgcHTMLComponent_Chart
- WebSocket or SSE

Integrating Third-Party JavaScript Libraries

sgcHTML renders complete pages on the server, but it stays open to the browser. Any JavaScript library (D3.js, Chart.js, a DevExtreme Grid or Chart, a calendar, a map) drops in with three small hooks, and can be wrapped in a reusable component.

Introduction

The built-in components cover charts, grids, forms, dashboards and much more, and several of them already wrap a JavaScript library internally. When you want a specific library of your own, you are never boxed in. Every browser library needs the same three things on the page, and **TsgcHTMLComponent_Site** exposes one member for each.

Member	Purpose
Custom-Head	Injects markup into the page `<head>`, the natural place for a `<script>` or `<link>` that loads the library.
AddContent	Emits the target element (a `<div>`, `<canvas>`, etc.) into the page body.
BodyEnd-HTML	Runs your initialization code right before `</body>`, after the library has loaded.

The Three Hooks

Here is a complete D3.js example: load the library, drop in a container, then let D3 draw into it.

```
#include "sgcHTML_Component_Site.hpp"

TsgcHTMLComponent_Site *oSite = new TsgcHTMLComponent_Site(NULL);
try
{
    oSite->Title = "Sales Dashboard";

    // 1) Load the library (a public CDN, or a file you serve yourself)
    oSite->CustomHead = "<script src=\"https://d3js.org/d3.v7.min.js\"></script>";

    // 2) Emit the element your library renders into
    oSite->AddContent("<div id=\"chart\"></div>");

    // 3) Run your init code at the end of the body
    oSite->BodyEndHTML =
        String("<script>")
        + "  const data = [4, 8, 15, 16, 23, 42];"
        + "  d3.select(\"#chart\").selectAll(\"div\").data(data).enter()"
        + "    .append(\"div\").style(\"width\", d => (d * 12) + \"px\").text(d => d);"
        + "</script>";

    WebModule->Response = oSite->HTML;
}
__finally
{
    oSite->Free();
}
```

That is the whole pattern. Swap the URL and the init code, and the same three members carry Chart.js, a DevExtreme widget, a calendar, a map, or anything else. The rest of this topic shows Delphi for brevity; the C++Builder and .NET code is identical in shape.

CDN or Self-Hosted

The example above points at a public CDN, which is the quickest way to try something. For an intranet application, an offline deployment, or when you want to pin an exact version, serve the library file yourself from the sgc server and reference it with a local path. Several built-in components already do this: the bundled **TsgcHTMLComponent_Chart** loads Chart.js from a local `/chart.umd.min.js` path rather than a CDN, so the page has no outbound dependency.

If you prefer typed nodes to a raw string, the template layer exposes `HeadNodes` and `BodyEndNodes` lists with `AddScript` and `AddStylesheet` helpers, plus **TsgcHTMLScript** (with `Src`, `Code`, `Defer`, `Async`) and **TsgcHTMLStylesheet** nodes. Raw strings and typed nodes are interchangeable.

Example: Chart.js

Load the library, emit a `<canvas>`, then construct the chart.

If you only want charts and do not need a specific library, note that sgcHTML already ships a **TsgcHTMLComponent_Chart** component that is Chart.js wired up for you, with a `CustomOptions` property that passes any raw Chart.js configuration straight through. Drive the library yourself only when you need full control.

Example: DevExtreme Grid

A heavier commercial suite works the same way. The DataGrid needs its stylesheet and its bundle loaded, an element to mount on, and a configuration object.

You can serialize a dataset to JSON on the server and inline it as shown, or point the grid at an endpoint your sgc server exposes and let it load live. A ready DevExtreme Grid sample that feeds a grid from an sgc WebSocket server ships in the demos folder.

Wrap It in a Reusable Component

Passing raw strings around is fine for a one-off. When you use the same library on page after page, promote it to a component: subclass **TsgcHTMLComponent** and override `GetHTML` to return your own markup plus the library glue. Everywhere else you then set a property and read `HTML`, exactly like a built-in component. Add a matching `GetCSS` override if the library needs styles, which sgcHTML collects and de-duplicates across the page.

```
class TsgcHTMLComponent_D3Bars : public TsgcHTMLComponent
{
private:
    String FData;   // a JSON array, e.g. "[4,8,15,16,23,42]"
__published:
    __property String Data = {read = FData, write = FData};
public:
    virtual String __fastcall GetHTML();
};

String __fastcall TsgcHTMLComponent_D3Bars::GetHTML()
{
    return String("<div id=\"d3bars\"></div>")
        + "<script src=\"https://d3js.org/d3.v7.min.js\"></script>"
```

```
        + "<script>  const data = " + FData + ";"
        + "  d3.select(\"#d3bars\").selectAll(\"span\").data(data).enter()"
        + "    .append(\"span\").style(\"padding-right\", d => d + \"px\").text(d => d);"
        + "</script>";
}
```

This is not a special case, it is how a number of the shipped components are built. Each of the following loads its library and emits its init code through exactly these hooks:

Component	Wrapped library
TsgcHTMLComponent_Map	Leaflet
TsgcHTMLComponent_RichEditor	Quill
TsgcHTMLComponent_KanbanBoard	SortableJS
TsgcHTMLComponent_Chart	Chart.js

Bundled htmx Extensions

Besides htmx itself and its WebSocket extension, sgcHTML embeds the following htmx 2 extensions in its resources. The engine serves each one from the local path shown, with the same content type as `/htmx-ext-ws.min.js`, so a page can load them with no CDN. The page template does not reference them by default.

Library	Version	License	Local path	URL
idiomorph (Idiomorph plus the `morph` htmx extension)	0.7.4	0BSD	`/idiomorph-ext.min.js`	https://github.com/bigskysoftware/idiomorph
htmx-ext-head-support	2.0.5	0BSD	`/htmx-ext-head-support.min.js`	https://htmx.org/extensions/head-support/
htmx-ext-sse	2.2.4	0BSD	`/htmx-ext-sse.min.js`	https://htmx.org/extensions/sse/

Related Topics

- TsgcHTMLComponent — the base component and the `GetHTML` extension point.
- Nodes — the node and composition layer (TsgcHTMLScript, TsgcHTMLStylesheet, raw markup).
- TsgcHTMLComponent_Chart — the built-in Chart.js component.

TsgcHTMLComponent_Chart

TsgcHTMLComponent_Chart renders a Chart.js chart (line, bar, area, spline, pie, doughnut, radar, pareto, mixed and more) or a server-side SVG chart (funnel, waterfall, bullet, radial bar) from data you supply directly or load from a dataset.

Introduction

TsgcHTMLComponent_Chart emits an HTML `<canvas>` element plus the Chart.js configuration script needed to draw it. You choose the chart type, add one or more axis labels, then add one or more datasets (a label plus a list of numeric values). The component builds the Chart.js JSON for you and returns the complete markup through its inherited **HTML** property.

The Chart.js library is referenced from a local path (`/chart.umd.min.js`) that the sgcHTML engine serves from an embedded resource, so no Content Delivery Network is required at runtime.

Quick Start

1. Drop a **TsgcHTMLComponent_Chart** component on the form (or create it at runtime).

2. Set the **ChartType**, add labels with **AddLabel** and a dataset with **AddDataset**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Chart *oChart = new TsgcHTMLComponent_Chart(NULL);
try
{
  oChart->ChartType = ctBar;
  oChart->Title = "Quarterly Revenue";
  oChart->AddLabel("Q1");
  oChart->AddLabel("Q2");
  oChart->AddLabel("Q3");
  oChart->AddDataset("Revenue", OPENARRAY(double, (1200, 1900, 1500)),
    "#7C3AED", "rgba(124,58,237,.25)", true);
  html = oChart->HTML;
}
__finally
{
  delete oChart;
}
```

Chart Type and Appearance

What it does: *ChartType* selects the Chart.js chart kind. *Title*, *Width*, *Height*, *ShowLegend*, *Responsive*, *Stacked* and *PointRadius* tune how the chart looks. *CustomOptions* lets you append a raw fragment of Chart.js options JSON for anything not exposed as a property.

Property	Type	De-fault	Description

ChartType	TsgcHTMLChartType	ctLine	Chart kind. Chart.js: ctLine, ctBar, ctPie, ctDoughnut, ctRadar, ctPolarArea, ctBubble, ctScatter, ctArea, ctHorizontalBar, ctStepLine, ctSpline, ctStackedBar100, ctMixed, ctPareto. Server-side SVG: ctFunnel, ctWaterfall, ctBullet, ctRadialBar.
Title	string	''	Optional title drawn above the chart.
Width	string	''	Canvas width in pixels. Empty lets the chart fill its container.
Height	string	'400'	Canvas height in pixels.
ShowLegend	Boolean	True	Shows or hides the dataset legend.
Responsive	Boolean	True	When True the chart resizes with its container.
Stacked	Boolean	False	Stacks datasets on the X and Y axes (bar/line charts).
PointRadius	Integer	3	Point radius used by scatter and bubble charts.
CustomOptions	string	''	Raw Chart.js options JSON appended to the generated options object.
ChartID	string	'sgcChart'	The id of the generated <canvas> element. Use a unique id when several charts share a page.

Data

What it does: A chart has a list of axis labels and one or more datasets. Add labels with **AddLabel**, add a dataset with **AddDataset**, and clear everything with **ClearData**. Each dataset is a **TsgcHTMLChartDataset** exposing *Label_*, *Data*, *BackgroundColor*, *BorderColor*, *Fill* and *BorderWidth*. You can also bind the chart to a dataset with **LoadFromDataSet** or by assigning *DataSource*.

```
// One label column, two value columns -> two datasets
String fields[2] = { "Sales", "Returns" };
oChart->LoadFromDataSet(qryMonths, "Month", OPENARRAY(String, ("Sales", "Returns")));
```

What's New

The chart component gained new chart types, per dataset types and multiple axes, value formatting, themes, PNG export, click drill-down and live updates pushed from the server. All the new properties default to the previous behaviour: a chart that does not set them produces exactly the same HTML as before.

The runtime demo **60.HTML\01.RunTime\15.Reports** (sgcReportsServer, *Analytics* page) shows most of the features described below: the new chart types, data labels, themes, PNG export, click drill-down into a grid, a live chart fed by **GetLiveAppendScript** and a candlestick chart with indicators and live ticks.

New Chart Types

ChartType has new values. Some are drawn by Chart.js in the browser, others are rendered on the server as an inline SVG because Chart.js has no equivalent. The SVG types emit no script at all, fill the width of their container and still use *Title*, *ShowLegend* (radial bar), *ShowDataLabels*, *DataLabelFormat*, *TooltipFormat* and the dataset colours.

Value	Rendering	Description
ctArea	Chart.js	Line chart with every dataset filled.
ctHorizontal-Bar	Chart.js	Horizontal bars.
ctStepLine	Chart.js	Stepped line.
ctSpline	Chart.js	Smooth curved line.
ctStacked-Bar100	Chart.js	Stacked bars where each label totals 100% (values converted to percentages on the server).
ctMixed	Chart.js	Bar chart whose datasets choose their own type (bars plus lines).
ctPareto	Chart.js	Bars sorted in descending order plus a cumulative percentage line on a right axis.
ctFunnel	SVG	Funnel stages with values and conversion percentage against the first stage.
ctWaterfall	SVG	Running total of increases and decreases, with an optional total bar.
ctBullet	SVG	Actual value bar, target marker and qualitative ranges per label.
ctRadialBar	SVG	One ring per label, filled in proportion to the value.

Mixed Charts and Multiple Axes

Every dataset has a *ChartType* (`cdtDefault`, `cdtLine`, `cdtBar`, `cdtArea`, `cdtScatter`, `cdtBubble`), a *YAxisID* and a *PointColors* list (one colour per point, bar or slice). See Datasets.

The Axes collection configures the scales: title, position, linear, logarithmic or category type, Min and Max, BeginAtZero, grid lines, stacking and a tick format. Add an axis with a new id (for example `y2`) and point a dataset to it with YAxisID to draw two value scales.

Value Format Syntax

DataLabelFormat, TooltipFormat and the *TickFormat* of an axis share one simple syntax: the text `{v}` is replaced by the value, in the browser, and the rest of the text is kept as is. Every occurrence of `{v}` is replaced.

A format without `{v}` is appended to the value as a suffix, so `%` is the same as `{v}%`. An empty format shows the bare value. The value itself is not rounded or localized; round it on the server when you add the data.

Format	Value	Text shown
`${v}`	1200	$1200
`{v} EUR`	1200	1200 EUR
`%`	45	45%
`{v} k ({v})`	12	12 k (12)
empty	1200	1200

Data Labels, Tooltips and Themes

ShowDataLabels draws each value on the chart (above bars and points, in the centre of slices), formatted with DataLabelFormat. The labels come from a small plugin that belongs to this chart only. TooltipFormat formats the tooltip value and keeps the dataset label as a prefix.

Theme selects the colours of the texts and grid lines: `chtLight` (Chart.js defaults), `chtDark` for dark pages, or `chtAuto`, which follows the Bootstrap `data-bs-theme` attribute of the page, or the `prefers-color-scheme` of the browser.

PNG Export

ShowExportPNG adds a small button below the chart that downloads it as a PNG image, with no server round trip. ExportPNGText sets the caption and ExportFileName the file name (`.png` is appended). Available for the Chart.js types.

Click Drill-down

Set ClickURL and a click on a point, bar or slice posts the point to the server through htmx: `chart`, `dataset`, `datasetLabel`, `index`, `label` and `value`. The answer is swapped into ClickTarget with ClickSwap (default `innerHTML`). Without a ClickTarget the answer is not swapped, which suits a server that replies with out-of-band fragments. htmx must be loaded on the page, as it is on the sgcHTMX engine pages.

Live Updates (Server Push)

A chart already shown in the browser can be updated without rendering it again. GetLiveAppendScript returns a fragment that appends a label and one value per dataset; GetLiveReplaceScript returns a fragment that replaces the labels and/or the data of one dataset. With LiveMaxPoints greater than zero the oldest points are dropped, so the chart keeps a sliding window.

Both methods return `<div id="<ChartID>" hx-swap-oob="beforeend">` with a small script that runs once, updates the chart through `Chart.getChart(ChartID)` and removes itself. Only ChartID and LiveMaxPoints are used, so a temporary instance with the same ChartID as the rendered chart is enough. Push the fragment to every connected browser with `BroadcastFragment` of the sgcHTMX engine (or to one client with `PushFragment`), or return it from an htmx polling request.

Output

The inherited **HTML** property returns a `<div>` wrapping the `<canvas>`, the Chart.js `<script>` reference and an inline initialization script. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 29 properties available on the component.
- Methods — 6 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Gauge

TsgcHTMLComponent_Gauge renders a semicircular SVG gauge that shows a single value between a minimum and maximum, with a color that changes as the value crosses configurable thresholds. It can also draw a 90, 180 or 270 degree scale with ticks, colored ranges and a needle, and update live.

Introduction

TsgcHTMLComponent_Gauge emits a self-contained `<svg>` drawing of a half-circle gauge. You set the **Value** together with the **MinValue** and **MaxValue** range; the component normalizes the value and fills the arc accordingly. A title, a unit caption and the min/max end labels are drawn as SVG text. The complete markup is returned through the inherited **HTML** property.

The gauge is drawn entirely with inline SVG, so it needs no JavaScript library at runtime. The arc color follows the value: below **ThresholdMid** it uses the low color, at or above **ThresholdHigh** it uses the high color, and in between it uses the mid color. Each color can be given either as an explicit CSS string (**ColorLow**, **ColorMid**, **ColorHigh**) or as a Bootstrap-style theme color (**ColorLowStyle**, **ColorMidStyle**, **ColorHighStyle**).

Quick Start

1. Drop a **TsgcHTMLComponent_Gauge** component on the form (or create it at runtime).

2. Set the **MinValue**, **MaxValue** and **Value**, and optionally a **Title** and **Unit_**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Gauge *oGauge = new TsgcHTMLComponent_Gauge(NULL);
try
{
  oGauge->Title = "CPU Load";
  oGauge->MinValue = 0;
  oGauge->MaxValue = 100;
  oGauge->Value = 72;
  oGauge->Unit_ = "%";
  html = oGauge->HTML;
}
__finally
{
  delete oGauge;
}
```

Value and Range

What it does: *Value* is the reading to display; *MinValue* and *MaxValue* set the range the arc spans. The component clamps the normalized value between the two ends, so a value outside the range simply pins the arc full or empty. *Title* is drawn as a caption below the gauge and *Unit_* is drawn under the value.

Property	Type	Default	Description

Value	Double	0	The reading drawn on the gauge.
MinValue	Double	0	The value at the left end of the arc.
MaxValue	Double	100	The value at the right end of the arc.
Title	string	''	Caption drawn below the gauge.
Unit_	string	''	Unit text drawn under the value (for example %).
GaugeID	string	''	The id of the wrapping ; falls back to sgcGauge.
Width	Integer	200	Overall width of the gauge in pixels.
SvgViewBox	string	'0 0 200 120'	The SVG viewBox used to scale the drawing.

Thresholds and Colors

What it does: The arc color tracks the value. Below *ThresholdMid* the low color is used, at or above *ThresholdHigh* the high color, and in between the mid color. Each color can be set as an explicit CSS string (*ColorLow*, *ColorMid*, *ColorHigh*); when the string is empty the matching theme color (*ColorLowStyle*, *ColorMidStyle*, *ColorHighStyle*) is used instead. The background arc and the labels follow the same string-or-theme pattern.

Property	Type	Default	Description
ThresholdMid	Double	50	At or above this value the mid color is used.
ThresholdHigh	Double	80	At or above this value the high color is used.
ColorLow	string	''	Explicit CSS color for the low band (overrides ColorLowStyle).
ColorMid	string	''	Explicit CSS color for the mid band (overrides ColorMidStyle).
ColorHigh	string	''	Explicit CSS color for the high band (overrides ColorHighStyle).
ColorLowStyle	TsgcHTMLColor	hcSuccess	Theme color used for the low band when ColorLow is empty.
ColorMidStyle	TsgcHTMLColor	hcWarning	Theme color used for the mid band when ColorMid is empty.
ColorHighStyle	TsgcHTMLColor	hcDanger	Theme color used for the high band when ColorHigh is empty.
BackgroundArcColor	string	''	Explicit CSS color of the unfilled background arc.

BackgroundArcColorStyle	TsgcHTMLColor	hcLight	Theme color of the background arc when BackgroundArcColor is empty.
LabelColor	string	''	Explicit CSS color of the unit and min/max labels.
LabelColorStyle	TsgcHTMLColor	hcSecondary	Theme color of the labels when LabelColor is empty.

Sizing

What it does: *ArcStrokeWidth* sets the thickness of the arc, and the three font-size properties control the value, unit and end-label text. *Width* and *SvgViewBox* together scale the whole drawing.

Property	Type	Default	Description
ArcStrokeWidth	Integer	12	Thickness of the gauge arc in SVG units.
ValueFontSize	Integer	24	Font size of the value text.
UnitFontSize	Integer	12	Font size of the unit text.
LabelFontSize	Integer	10	Font size of the min/max end labels.

Scale, Ranges and Needle

What it does: *ArcAngle* switches the gauge from the original semicircle (0, the default, whose output is unchanged) to the scale renderer with a sweep of 90, 180 or 270 degrees. The scale renderer adds tick marks and labels (*ShowTicks*, *MajorTicks*, *MinorTicks*, *ShowTickLabels*, *TickColor*), colored bands (*Ranges*, items of type TsgcHTMLInstrumentRange), a needle (*NeedleStyle*, *NeedleColor*) and a smooth transition between values (*Animate*, off for users who ask for reduced motion). The value arc takes the color of the first range that contains the value, otherwise the threshold colors. These properties have no effect while ArcAngle is 0.

Property	Type	Default	Description
ArcAngle	Integer	0	Sweep of the scale: 0 (legacy semicircle), 90, 180 or 270 degrees.
ShowTicks	Boolean	False	Draws major and minor ticks inside the scale.
MajorTicks	Integer	0	Number of major intervals; 0 means automatic.
MinorTicks	Integer	4	Minor ticks between two major ticks.
ShowTickLabels	Boolean	True	Prints the value of every major tick.

TickColor	string	''	CSS color of the ticks and tick labels; empty uses the label color.
Ranges	TsgcHTMLInstrumentRanges	empty collection	Colored bands of the scale (StartValue, EndValue, Color, Caption).
NeedleStyle	TsgcHTMLGaugeNeedleStyle	gnsNone	Draws a needle pointing at the value.
NeedleColor	string	''	CSS color of the needle; empty uses the dark theme color.
Animate	Boolean	False	Animates the arc and needle when the value changes.

```
oGauge->GaugeID = "boilerTemp";
oGauge->ArcAngle = 270;
oGauge->ShowTicks = true;
oGauge->NeedleStyle = gnsTriangle;
TsgcHTMLInstrumentRange *oRange = oGauge->Ranges->Add();
oRange->StartValue = 80;
oRange->EndValue = 100;
oRange->Color = "#dc3545";
oRange->Caption = "Danger";
```

Live Updates

What it does: *GetLiveValueFragment* sets *Value* and returns the whole gauge with `hx-swap-oob="outerHTML"` on its root, whose id is *GaugeID* (or sgcGauge). Broadcast it with **TsgcHTMX_Engine_Server.BroadcastFragment** and the gauge is replaced in every browser that shows it. Give the gauge an id that starts with a letter. See Instruments for the full flow, and **Demos\60.HTML\01.RunTime\20.Instruments** for a working control room.

```
HTMXServer->BroadcastFragment(oGauge->GetLiveValueFragment(vTemperature));
```

Output

The inherited **HTML** property returns a centered `<div>` containing the inline `<svg>` gauge and a small caption. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document. For a one-off gauge with no component instance, use the static **Build** class function.

Reference

- Properties — 34 properties available on the component.
- Methods — 2 public methods.

- Events — this component has no published events.

Instruments

Gauges, lamps, displays, operator controls, real-time trends and process diagrams that the server keeps live in the browser.

Introduction

The instrumentation components draw the panels of a control room, a test bench or a machine HMI as plain HTML and inline SVG. Each one is a server-side generator like every other sgcHTML component: you set properties, read **HTML** and serve it. No JavaScript library and no image is needed. What makes them instruments is that each one can also produce a small **live update** that the sgcHTMX engine pushes to every browser showing the page, so the needles move, the lamps light and the trends scroll with no page reload and no polling.

Every instrument draws an accessible SVG (`role="img"` with an `aria-label` describing the reading, or the matching ARIA role on the input controls), and every animation is switched off for users whose browser asks for reduced motion.

The Family

Displays show a reading.

Component	What it draws	Live update
TsgcHTMLComponent_Gauge	Semicircle gauge, or a 90, 180 or 270 degree scale with ticks, ranges and a needle.	GetLiveValueFragment
TsgcHTMLComponent_LinearGauge	Straight bar, thermometer or tank, horizontal or vertical.	GetLiveValueFragment
TsgcHTMLComponent_LED	On/off lamp, round or square, with glow and blink.	GetLiveStateFragment
TsgcHTMLComponent_LEDBar	Segmented LED bar graph.	GetLiveValueFragment
TsgcHTMLComponent_SegmentDisplay	7-segment or 14-segment LED text.	GetLiveTextFragment
TsgcHTMLComponent_Annunciator	Alarm annunciator panel of tiles with operator acknowledgement.	GetLiveTileFragment
TsgcHTMLComponent_Odometer	Mechanical counter with rolling digit wheels.	GetLiveValueFragment
TsgcHTMLComponent_VUMeter	Audio level meter in dB, needle or bar, with peak hold.	GetLiveValueFragment

TsgcHTMLComponent_AnalogClock	Analog clock, static or ticking, for any UTC offset.	GetLiveTimeFragment
TsgcHTMLComponent_Compass	Compass or heading indicator with a target marker.	GetLiveValueFragment

Inputs let the operator act. They post their value to a URL (or send it over the WebSocket) and can be updated from the server like a display.

Component	What it draws
TsgcHTMLComponent_Knob	Rotary knob.
TsgcHTMLComponent_ToggleSwitch	Toggle or rocker switch.
TsgcHTMLComponent_PushButton	Push button with an indicator lamp, latching or momentary.
TsgcHTMLComponent_NumericStepper	Numeric entry with step buttons.

Real-time charts draw a stream of samples on a canvas in the browser. They are fed with small push scripts instead of whole fragments.

Component	What it draws
TsgcHTMLComponent_StripChart	Scrolling strip chart recorder with several pens.
TsgcHTMLComponent_Oscilloscope	Oscilloscope screen with channels and a trigger.

SCADA and **MQTT binding** complete the family.

Component	What it does
TsgcHTMLComponent_SCADAPanel	Process diagram of tanks, valves, pumps, motors and sensors joined by pipes with animated flow; every symbol and pipe can be updated on its own.
TsgcHTMLInstrumentsBinder	Binds topics to instruments without any transport: call ProcessMessage from any source (AMQP, Kafka, a WebSocket, a simulator) and the bound instruments update and broadcast their fragments. In .NET the OnBroadcast event sends them through ASP.NET Core.
TsgcHTMLInstrumentsMQTTBinder	Binds MQTT topics to instruments: every message received on a topic updates the bound instrument and broadcasts the fragment, with no code.

How Live Updates Work

1. Render the page once. Place the instruments in a page served through the sgcHTMX engine (TsgcHTMX_Engine_Server), so the browser opens the sgcHTMX WebSocket bridge. Give each instrument its id property (GaugeID, LEDID, CompassID and so on).

2. Ask the instrument for a fragment. When a reading changes, call the **GetLive*Fragment** method of the instrument (GetLiveValueFragment, GetLiveStateFragment, GetLiveTextFragment, GetLiveTimeFragment or GetLiveTileFragment). It stores the new value in the component and returns the instrument markup with `hx-swap-oob="outerHTML"` on its root element: an **out-of-band fragment**. The Annunciator and the SCADA panel return a single tile, symbol or pipe, so only that part is sent.

3. Broadcast it. Send the string with BroadcastFragment (every client, or only the clients of a channel) or PushFragment (one client). htmx finds the element with the same id in the page and replaces it. Several fragments can be concatenated and sent in one push.

```
// page render
oGauge->GaugeID = "boilerTemp";
oGauge->ArcAngle = 270;
oGauge->Animate = true;
oLED->LEDID = "heaterOn";
...
// every time the process changes
HTMXServer->BroadcastFragment(
  oGauge->GetLiveValueFragment(vTemperature) +
  oLED->GetLiveStateFragment(vHeaterOn));
```

A few rules keep the updates reliable:

- **Same id for render and update.** The fragment replaces the element whose id matches its root id. Use the same component, or at least the same id property, for the first render and for every update.
- **Ids never start with a digit.** htmx locates the element to swap with a CSS id selector, and an id that starts with a digit is not a valid selector, so the swap silently does nothing. Start every instrument id with a letter (`tank1`, not `1tank`). Inner elements are named after the root id (`<id>-value`, `<id>-needle` and so on).
- **One id per instrument on a page.** When an id property is empty the default id of the class is used (sgcGauge, sgcLED ...), so two instruments of the same class on one page need their own ids.
- **Reduced motion.** Transitions (Animate on the gauges and compass, rolling odometer wheels) and blinking (LED, Annunciator) are wrapped in a `prefers-reduced-motion` media query: users who ask the browser for reduced motion see the new value straight away, with no animation.
- **Real-time charts are different.** The StripChart and the Oscilloscope keep their samples in the browser; you append samples with their push script methods rather than replacing the chart with a fragment.

With the TsgcHTMLInstrumentsMQTTBinder the same flow runs by itself from MQTT messages: each binding maps a topic (optionally a JSON path inside the payload) to an instrument, and the binder calls the right live method and broadcasts the fragment through the HTMX server. The MQTT binder descends from TsgcHTMLInstrumentsBinder, which does the same work for messages from any source: pass each one to **ProcessMessage**.

Demo

Demos\60.HTML\01.RunTime\20.Instruments is a dark control room for a simulated boiler loop: a pump feeds a boiler, a heater warms the water and an outlet valve drains it into a tank. A background thread advances the process and pushes the new values to every connected browser twice a second, and the Controls page posts every operator action back to the server. **Demos\60.HTML\02.DesignTime\20.Instruments** is the same demo built on forms with the Design Surface.

TsgcHTMLComponent_LinearGauge

TsgcHTMLComponent_LinearGauge renders a straight SVG gauge, as a bar, a thermometer or a tank, with a tick scale, colored ranges and a value readout.

Introduction

TsgcHTMLComponent_LinearGauge draws a single value between **MinValue** and **MaxValue** along a straight scale, horizontal or vertical. **GaugeStyle** chooses the look: a plain bar, a thermometer with a bulb, or a tank whose liquid level rises with the value.

The scale has major and minor ticks with labels, and the **Ranges** collection draws colored bands next to it. The fill color is the color of the first range that contains the value, otherwise **BarColor**, otherwise the primary theme color. With **Animate** the fill glides to a new value (disabled when the browser asks for reduced motion).

Quick Start

1. Drop a **TsgcHTMLComponent_LinearGauge** on the form (or create it at runtime).

2. Set **MinValue**, **MaxValue**, **Value** and choose a **GaugeStyle** and **Orientation**.

3. Read the **HTML** property and serve it, then call **GetLiveValueFragment** to push new readings.

```
TsgcHTMLComponent_LinearGauge *oGauge = new TsgcHTMLComponent_LinearGauge(NULL);
try
{
  oGauge->LinearGaugeID = "tankLevel";
  oGauge->Title = "Tank level";
  oGauge->GaugeStyle = lgsTank;
  oGauge->Orientation = loVertical;
  oGauge->Value = 64;
  oGauge->Unit_ = "%";
  html = oGauge->HTML;
}
__finally
{
  delete oGauge;
}
```

Value and Range

Property	Type	Default	Description
Value	Double	0	The reading shown by the gauge.
MinValue	Double	0	The value at the start of the scale.
MaxValue	Double	100	The value at the end of the scale.

Unit_	string	'' (empty)	Unit text appended to the value readout.

Identity

Property	Type	Default	Description
Title	string	'' (empty)	Caption drawn below the gauge.
LinearGaugeID	string	'' (empty)	The id of the root element; falls back to sgcLinearGauge.

Appearance

Property	Type	Default	Description
Orientation	TsgcHTMLLinearOrientation	loHorizontal	Horizontal or vertical scale.
GaugeStyle	TsgcHTMLLinearGaugeStyle	lgsBar	Bar, thermometer or tank look.
DrawingWidth	Integer	0	Drawing width in pixels; 0 means automatic.
DrawingHeight	Integer	0	Drawing height in pixels; 0 means automatic.
BarColor	string	'' (empty)	CSS color of the fill; empty uses the primary color.
ShowValue	Boolean	True	Shows the value readout.
Animate	Boolean	False	Animates the fill when the value changes.

Scale and Ranges

Property	Type	Default	Description
ShowTicks	Boolean	True	Draws the major and minor ticks.

MajorTicks	Integer	0	Number of major intervals; 0 means automatic.
MinorTicks	Integer	4	Minor ticks between two major ticks.
ShowTickLabels	Boolean	True	Prints the value of every major tick.
Ranges	TsgcHTMLInstrumentRanges	empty collection	Colored bands of the scale (StartValue, EndValue, Color, Caption).

Live Updates

Call **GetLiveValueFragment** whenever the reading changes. It stores the new value and returns the whole instrument with `hx-swap-oob="outerHTML"` on its root element. Broadcast that string with the sgcHTMX engine and every browser that shows the page swaps the instrument in place. See Instruments for the full flow.

Output

The inherited **HTML** property returns a `<div>` that carries the id, an inline `<svg>` with `role="img"` and an `aria-label` describing the reading, and an optional small title. No JavaScript library is needed. For a one-off instrument with no component instance, use the static **Build** class function.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) is a control room for a simulated boiler loop that uses the instrumentation components together and keeps them live over the sgcHTMX WebSocket bridge.

Reference

- Properties — 18 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_LED

TsgcHTMLComponent_LED renders an on/off indicator lamp, round or square, with an optional glow, blink and caption.

Introduction

TsgcHTMLComponent_LED is the simplest instrument: a lamp that is either on or off. When **State** is True the lamp uses **OnColor** with a soft glow; when False it uses **OffColor**. **Blink** makes a lit lamp flash, which is useful for alarms. The blink animation is disabled for users whose browser asks for reduced motion.

Quick Start

1. Drop a **TsgcHTMLComponent_LED** on the form (or create it at runtime).

2. Set **State**, **Caption** and, if needed, **OnColor**.

3. Read the **HTML** property, then call **GetLiveStateFragment** when the state changes.

```
TsgcHTMLComponent_LED *oLED = new TsgcHTMLComponent_LED(NULL);
try
{
  oLED->LEDID = "pumpRunning";
  oLED->Caption = "Pump";
  oLED->State = true;
  html = oLED->HTML;
}
__finally
{
  delete oLED;
}
```

State

Property	Type	Default	Description
State	Boolean	False	True lights the lamp.
Blink	Boolean	False	Makes the lit lamp flash.

Appearance

Property	Type	Default	Description
OnColor	string	'' (empty)	CSS color of the lit lamp; empty uses the success green.
OffColor	string	'' (empty)	CSS color of the unlit lamp; empty uses a dimmed grey.
Shape	TsgcHTMLLED-Shape	lsCircle	Round or square lamp.
Size	Integer	16	Lamp size in pixels.

Identity

Property	Type	Default	Description
Caption	string	'' (empty)	Text shown next to the lamp.
LEDID	string	'' (empty)	The id of the root element; falls back to sgcLED.

Live Updates

Call **GetLiveStateFragment** whenever the reading changes. It stores the new value and returns the whole instrument with `hx-swap-oob="outerHTML"` on its root element. Broadcast that string with the sgcHTMX engine and every browser that shows the page swaps the instrument in place. See Instruments for the full flow.

Output

The inherited **HTML** property returns a `<div>` that carries the id, an inline `<svg>` with `role="img"` and an `aria-label` describing the reading, and an optional small title. No JavaScript library is needed. For a one-off instrument with no component instance, use the static **Build** class function.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) is a control room for a simulated boiler loop that uses the instrumentation components together and keeps them live over the sgcHTMX WebSocket bridge.

Reference

- Properties — 8 properties available on the component.

- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_LEDBar

TsgcHTMLComponent_LEDBar renders a segmented LED bar graph, the classic level meter made of individual lamps.

Introduction

TsgcHTMLComponent_LEDBar splits the scale **MinValue**..**MaxValue** into **SegmentCount** lamps and lights as many of them as the value reaches (the ratio times the count, rounded). Unlit segments are drawn with **OffColor**, or with their own color dimmed.

Segment colors come from the **Ranges** collection: each segment takes the color of the range that contains the value at its centre. With no range, the segments are green in the first third, yellow in the second and red in the last.

Quick Start

1. Drop a **TsgcHTMLComponent_LEDBar** on the form (or create it at runtime).

2. Set **MinValue**, **MaxValue**, **Value** and **SegmentCount**.

3. Read the **HTML** property, then call **GetLiveValueFragment** for new readings.

```
TsgcHTMLComponent_LEDBar *oBar = new TsgcHTMLComponent_LEDBar(NULL);
try
{
  oBar->LEDBarID = "signal";
  oBar->Title = "Signal";
  oBar->SegmentCount = 12;
  oBar->Value = 70;
  html = oBar->HTML;
}
__finally
{
  delete oBar;
}
```

Value and Range

Property	Type	Default	Description
Value	Double	0	The reading shown by the bar.
MinValue	Double	0	The value at the start of the bar.
MaxValue	Double	100	The value at the end of the bar.

Identity

Property	Type	Default	Description
Title	string	'' (empty)	Caption drawn below the bar.
LEDBarID	string	'' (empty)	The id of the root element; falls back to sgcLEDBar.

Appearance

Property	Type	Default	Description
SegmentCount	Integer	10	Number of lamps (1 to 100).
Orientation	TsgcHTMLLinearOrientation	loHorizontal	Horizontal or vertical bar.
DrawingWidth	Integer	0	Drawing width in pixels; 0 means automatic.
DrawingHeight	Integer	0	Drawing height in pixels; 0 means automatic.
OffColor	string	'' (empty)	CSS color of unlit segments; empty dims the segment color.
Ranges	TsgcHTMLInstrumentRanges	empty collection	Colored bands of the scale (StartValue, EndValue, Color, Caption).

Live Updates

Call **GetLiveValueFragment** whenever the reading changes. It stores the new value and returns the whole instrument with `hx-swap-oob="outerHTML"` on its root element. Broadcast that string with the sgcHTMX engine and every browser that shows the page swaps the instrument in place. See Instruments for the full flow.

Output

The inherited **HTML** property returns a `<div>` that carries the id, an inline `<svg>` with `role="img"` and an `aria-label` describing the reading, and an optional small title. No JavaScript library is needed. For a one-off instrument with no component instance, use the static **Build** class function.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) is a control room for a simulated boiler loop that uses the instrumentation components together and keeps them live over the sgcHTMX WebSocket bridge.

Reference

- Properties — 11 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_SegmentDisplay

TsgcHTMLComponent_SegmentDisplay renders text as a 7-segment or 14-segment LED display, with ghost segments and an optional italic slant.

Introduction

TsgcHTMLComponent_SegmentDisplay draws **Text** the way a digital panel meter does. The 7-segment type shows digits, - and space; the 14-segment type adds the letters A to Z (lowercase letters are shown as uppercase). A ., , or : is drawn as a dot or colon between digits and does not use a digit position.

Unlit segments are drawn faintly (ghost segments) so the display looks like real hardware. **Digits** fixes the number of positions: the text is right aligned and the leftmost characters are dropped when it does not fit.

Quick Start

1. Drop a **TsgcHTMLComponent_SegmentDisplay** on the form (or create it at runtime).

2. Set **Text**, **Digits** and the colors.

3. Read the **HTML** property, then call **GetLiveTextFragment** when the text changes.

```
TsgcHTMLComponent_SegmentDisplay *oDisplay = new TsgcHTMLComponent_SegmentDisplay(NULL);
try
{
  oDisplay->SegmentDisplayID = "pressure";
  oDisplay->Text = "12.75";
  oDisplay->Digits = 6;
  oDisplay->OnColor = "#ff3b30";
  oDisplay->BackgroundColor = "#111";
  html = oDisplay->HTML;
}
__finally
{
  delete oDisplay;
}
```

Content

Property	Type	Default	Description
Text	string	'' (empty)	The characters to display.
Digits	Integer	0	Number of digit positions; 0 fits the text.

Identity

Property	Type	Default	Description
Title	string	'' (empty)	Caption drawn below the display.
SegmentDisplayID	string	'' (empty)	The id of the root element; falls back to sgcSegmentDisplay.

Appearance

Property	Type	Default	Description
SegmentType	TsgcHTMLSegmentDisplayType	sdt7Segment	7-segment or 14-segment characters.
OnColor	string	'' (empty)	CSS color of lit segments; empty uses currentColor.
OffColor	string	'' (empty)	CSS color of ghost segments; empty dims OnColor.
BackgroundColor	string	'' (empty)	CSS background of the display; empty is transparent.
SkewAngle	Integer	0	Italic slant in degrees (-30 to 30).
DigitHeight	Integer	48	Digit height in pixels.

Live Updates

Call **GetLiveTextFragment** whenever the reading changes. It stores the new value and returns the whole instrument with `hx-swap-oob="outerHTML"` on its root element. Broadcast that string with the sgcHTMX engine and every browser that shows the page swaps the instrument in place. See Instruments for the full flow.

Output

The inherited **HTML** property returns a `<div>` that carries the id, an inline `<svg>` with `role="img"` and an `aria-label` describing the reading, and an optional small title. No JavaScript library is needed. For a one-off instrument with no component instance, use the static **Build** class function.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) is a control room for a simulated boiler loop that uses the instrumentation components together and keeps them live over the sgcHTMX WebSocket bridge.

Reference

- Properties — 10 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Annunciator

TsgcHTMLComponent_Annunciator renders an alarm annunciator panel: a grid of captioned tiles that light up by state and can be acknowledged by the operator.

Introduction

TsgcHTMLComponent_Annunciator reproduces the alarm window panel of a control room. Every item of **Tiles** is a TsgcHTMLAnnunciatorTile with a caption and a state: normal, advisory, warning or alarm. The tiles are laid out in a CSS grid of **Columns** columns and colored by state.

A tile that is not normal and not yet acknowledged is pending. With **Blink**, pending warning and alarm tiles flash (not for users who ask for reduced motion). When **AckURL** is set, a pending tile becomes a button: a click, Enter or Space posts `annunciator` and `tile` to that URL with htmx, so the server can set **Acknowledged** and push the tile back with **GetLiveTileFragment**.

Quick Start

1. Drop a **TsgcHTMLComponent_Annunciator** on the form (or create it at runtime).

2. Add tiles with `Tiles.Add`, giving each a **TileID** and **Caption**.

3. Read the **HTML** property; when a state changes, set it on the tile and broadcast **GetLiveTileFragment**.

```
TsgcHTMLComponent_Annunciator *oPanel = new TsgcHTMLComponent_Annunciator(NULL);
try
{
  oPanel->AnnunciatorID = "alarms";
  oPanel->Columns = 3;
  oPanel->AckURL = "/alarms/ack";
  TsgcHTMLAnnunciatorTile *oTile = oPanel->Tiles->Add();
  oTile->TileID = "highTemp";
  oTile->Caption = "HIGH TEMP";
  oTile->State = ansAlarm;
  oTile = oPanel->Tiles->Add();
  oTile->TileID = "lowFlow";
  oTile->Caption = "LOW FLOW";
  html = oPanel->HTML;
}
__finally
{
  delete oPanel;
}
```

Tiles

Property	Type	Default	Description
Tiles	TsgcHTMLAnnunciatorTiles	empty collection	The tiles of the panel.

Columns	Integer	4	Number of grid columns (1 to 24).

Behavior

Property	Type	Default	Description
AnnunciatorID	string	'' (empty)	The id of the root element; falls back to sgcAnnunciator.
Blink	Boolean	True	Pending warning and alarm tiles flash.
AckURL	string	'' (empty)	URL a pending tile posts to when the operator acknowledges it.

Colors

Property	Type	Default	Description
NormalColor	string	'' (empty)	CSS color of normal tiles; empty is dark grey.
AdvisoryColor	string	'' (empty)	CSS color of advisory tiles; empty is blue.
WarningColor	string	'' (empty)	CSS color of warning tiles; empty is amber.
AlarmColor	string	'' (empty)	CSS color of alarm tiles; empty is red.

Tile States

State	Default color	Meaning
ansNormal	#343a40 (NormalColor)	Nothing to report. Never pending.
ansAdvisory	#0d6efd (AdvisoryColor)	Information for the operator. Pending until acknowledged, does not blink.
ansWarning	#ffc107 (WarningColor)	Abnormal condition. Blinks while pending.
ansAlarm	#dc3545 (AlarmColor)	Critical condition. Blinks while pending.

The state prefix is `ans` because `asWarning` already exists in TsgcHTMLAlertStyle.

Live Updates

Change a tile and call **GetLiveTileFragment** with its TileID. It returns only that tile, marked with `hx-swap-oob="outerHTML"`, so a broadcast swaps a single tile. See Instruments for the full flow.

Output

The inherited **HTML** property returns a `<div>` with the root id holding a CSS grid of tile `<div>` elements. Every tile carries `data-state` and an `aria-label` with its caption and state; tiles that are not buttons have `role="status"`.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) is a control room for a simulated boiler loop that uses the instrumentation components together and keeps them live over the sgcHTMX WebSocket bridge.

Reference

- Properties — 9 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.
- TsgcHTMLAnnunciatorTile — the tile item of the Tiles collection.

TsgcHTMLAnnunciatorTile

TsgcHTMLAnnunciatorTile is one tile of the TsgcHTMLComponent_Annunciator panel: a caption with an alarm state and an acknowledged flag.

Introduction

Tiles are the items of the **Tiles** collection (**TsgcHTMLAnnunciatorTiles**). Add them with `Tiles.Add`, read them with `Tiles[i]` and find one by id with `Tiles.FindTile(aTileID)`, which returns nil (null in C#) when no tile has that TileID. **Assign** copies the four published properties of another tile.

Properties

Property	Type	Default	Description
TileID	string	''	Identifies the tile. The tile element id is <AnnunciatorID>-<TileID> (reduced to safe id characters), or <AnnunciatorID>-tile<Index> when empty. It is also the `tile` value posted to AckURL and the argument of GetLiveTileFragment. Start it with a letter.
Caption	string	''	The text shown on the tile.
State	TsgcHTMLAnnunciatorState	ansNormal	ansNormal, ansAdvisory, ansWarning or ansAlarm. Selects the tile color.
Acknowledged	Boolean	False	True once the operator acknowledged the condition. A tile that is not normal and not acknowledged is pending: it can blink and, with AckURL, it is clickable.

Example

```
TsgcHTMLAnnunciatorTile *oTile = oPanel->Tiles->Add();
oTile->TileID = "highPressure";
oTile->Caption = "HIGH PRESS";
oTile->State = ansWarning;
oTile->Acknowledged = false;
```

TsgcHTMLComponent_Odometer

TsgcHTMLComponent_Odometer renders a mechanical counter whose digit wheels roll to the new value.

Introduction

TsgcHTMLComponent_Odometer shows a number on a row of digit wheels, like the distance counter of a car or a utility meter. **Digits** sets the integer wheels and **Decimals** the fraction wheels, which are drawn with the colors swapped.

Every wheel is positioned with a CSS translate, so after a live update the wheels roll to the new digits. A value larger than the wheels can hold wraps around like a real odometer, and a negative value is shown as 0.

Quick Start

1. Drop a **TsgcHTMLComponent_Odometer** on the form (or create it at runtime).

2. Set **Digits**, **Decimals** and **Value**.

3. Read the **HTML** property, then call **GetLiveValueFragment** for new counts.

```
TsgcHTMLComponent_Odometer *oCounter = new TsgcHTMLComponent_Odometer(NULL);
try
{
  oCounter->OdometerID = "energy";
  oCounter->Digits = 6;
  oCounter->Decimals = 1;
  oCounter->Unit_ = "kWh";
  oCounter->Value = 10423.7;
  html = oCounter->HTML;
}
__finally
{
  delete oCounter;
}
```

Value

Property	Type	Default	Description
Value	Double	0	The number shown on the wheels.
Digits	Integer	6	Number of integer wheels (1 to 15).
Decimals	Integer	0	Number of decimal wheels (0 to 3).
Unit_	string	” (empty)	Unit text shown after the wheels.

Appearance

Property	Type	Default	Description
OdometerID	string	'' (empty)	The id of the root element; falls back to sgcOdometer.
DigitColor	string	'' (empty)	CSS color of the digits; empty is white.
BackgroundColor	string	'' (empty)	CSS color of the wheels; empty is near black.

Live Updates

Call **GetLiveValueFragment** whenever the reading changes. It stores the new value and returns the whole instrument with `hx-swap-oob="outerHTML"` on its root element. Broadcast that string with the sgcHTMX engine and every browser that shows the page swaps the instrument in place. See Instruments for the full flow.

Output

The inherited **HTML** property returns a `<div>` that carries the id, an inline `<svg>` with `role="img"` and an `aria-label` describing the reading, and an optional small title. No JavaScript library is needed. For a one-off instrument with no component instance, use the static **Build** class function.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) is a control room for a simulated boiler loop that uses the instrumentation components together and keeps them live over the sgcHTMX WebSocket bridge.

Reference

- Properties — 7 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_VUMeter

TsgcHTMLComponent_VUMeter renders an audio VU meter in decibels, as a classic needle meter or as a bar, with optional peak hold.

Introduction

TsgcHTMLComponent_VUMeter shows a level in dB on the scale **MinValue**..**MaxValue** (-20 to +3 by default). The position follows the classic VU law, linear in voltage, so 0 dB sits well to the right of the centre, and the part above 0 dB is drawn in red.

MeterStyle chooses a needle meter (the needle swings from -45 to +45 degrees) or a horizontal bar. Every time **Value** is set, the highest value is remembered in **PeakValue**; with **PeakHold** the peak is drawn as a marker until **ResetPeak** is called.

Quick Start

1. Drop a **TsgcHTMLComponent_VUMeter** on the form (or create it at runtime).

2. Set **MeterStyle**, **PeakHold** and **Value** (in dB).

3. Read the **HTML** property, then call **GetLiveValueFragment** for every new level.

```
TsgcHTMLComponent_VUMeter *oMeter = new TsgcHTMLComponent_VUMeter(NULL);
try
{
  oMeter->VUMeterID = "vuLeft";
  oMeter->Title = "Left";
  oMeter->PeakHold = true;
  oMeter->Value = -3;
  html = oMeter->HTML;
}
__finally
{
  delete oMeter;
}
```

Value and Range

Property	Type	Default	Description
Value	Double	-20	The level in dB.
MinValue	Double	-20	The dB value at the start of the scale.
MaxValue	Double	3	The dB value at the end of the scale.

PeakValue	Double	-20	Highest Value since creation or the last ResetPeak (public, read only).

Appearance

Property	Type	Default	Description
MeterStyle	TsgcHTMLVUMeterStyle	vmsNeedle	Needle meter or bar.
PeakHold	Boolean	False	Draws a marker at PeakValue.
Title	string	'' (empty)	Caption drawn below the meter.
VUMeterID	string	'' (empty)	The id of the root element; falls back to sgcVUMeter.

Live Updates

Call **GetLiveValueFragment** whenever the reading changes. It stores the new value and returns the whole instrument with `hx-swap-oob="outerHTML"` on its root element. Broadcast that string with the sgcHTMX engine and every browser that shows the page swaps the instrument in place. See Instruments for the full flow.

Output

The inherited **HTML** property returns a `<div>` that carries the id, an inline `<svg>` with `role="img"` and an `aria-label` describing the reading, and an optional small title. No JavaScript library is needed. For a one-off instrument with no component instance, use the static **Build** class function.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) is a control room for a simulated boiler loop that uses the instrumentation components together and keeps them live over the sgcHTMX WebSocket bridge.

Reference

- Properties — 8 properties available on the component.
- Methods — 3 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_AnalogClock

TsgcHTMLComponent_AnalogClock renders an analog clock face with hour, minute and second hands, static or ticking in the browser.

Introduction

TsgcHTMLComponent_AnalogClock draws a clock face showing **Time**. When Time is 0 the clock shows the server time at render, or UTC plus **UTCOffset** minutes when **UseUTCOffset** is True, which is how a wall of clocks for several time zones is built.

With **Live** the page includes a small script that advances the hands every second in the browser, starting from the rendered time (or from the browser UTC clock plus UTCOffset when UseUTCOffset is True). The script stops by itself when the clock is removed from the page.

Quick Start

1. Drop a **TsgcHTMLComponent_AnalogClock** on the form (or create it at runtime).

2. Set **UseUTCOffset** and **UTCOffset** for a time zone, and **Live** for a ticking clock.

3. Read the **HTML** property and serve it.

```
TsgcHTMLComponent_AnalogClock *oClock = new TsgcHTMLComponent_AnalogClock(NULL);
try
{
  oClock->AnalogClockID = "clockTokyo";
  oClock->Title = "Tokyo";
  oClock->UseUTCOffset = true;
  oClock->UTCOffset = 540;
  oClock->Live = true;
  html = oClock->HTML;
}
__finally
{
  delete oClock;
}
```

Time

Property	Type	Default	Description
Time	TDateTime	0 (current time)	The time shown; 0 means the current time at render.
ShowSeconds	Boolean	True	Draws the second hand.
Live	Boolean	False	The browser advances the hands every second.
UseUTCOffset	Boolean	False	Shows UTC plus UTCOffset instead of local time.

UTCOffset	Integer	0	Minutes east of UTC.

Appearance

Property	Type	Default	Description
Numerals	TsgcHTMLClockNumerals	cknArabic	Arabic, Roman or no numerals on the face.
FaceColor	string	'' (empty)	CSS color of the face; empty is white.
HandColor	string	'' (empty)	CSS color of the hour and minute hands; empty uses currentColor.
SecondHandColor	string	'' (empty)	CSS color of the second hand; empty is red.
Title	string	'' (empty)	Caption drawn below the clock.
AnalogClockID	string	'' (empty)	The id of the root element; falls back to sgcAnalogClock.

Live Clocks

There are two ways to keep a clock current. **Live** runs a timer in the browser and needs no server traffic. **GetLiveTimeFragment** sets the time on the server and returns the clock as an out-of-band fragment to broadcast with the sgcHTMX engine, which is useful when the time comes from a device or a simulation. See Instruments.

Output

The inherited **HTML** property returns a `<div>` that carries the id, an inline `<svg>` with `role="img"` and an `aria-label` describing the reading, and an optional small title. No JavaScript library is needed. For a one-off instrument with no component instance, use the static **Build** class function.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) is a control room for a simulated boiler loop that uses the instrumentation components together and keeps them live over the sgcHTMX WebSocket bridge.

Reference

- Properties — 11 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Compass

TsgcHTMLComponent_Compass renders a compass or heading indicator with a rotating card or a rotating needle, and an optional target marker.

Introduction

TsgcHTMLComponent_Compass shows **Heading** in degrees, 0 being north and angles growing clockwise. With the default `csRotatingCard` style the card turns so the heading sits under the fixed lubber line at the top, like an aircraft heading indicator; with `csRotatingNeedle` the card is fixed and a needle points to the heading.

The card shows the eight compass points and a degree scale. **ShowTarget** adds a marker (a heading bug) at **TargetHeading**. With **Animate** the card or needle turns smoothly after a live update, always along the shortest path, so going from 350 to 10 degrees turns 20 degrees and not 340.

Quick Start

1. Drop a **TsgcHTMLComponent_Compass** on the form (or create it at runtime).

2. Set **Heading** and, if needed, **CompassStyle** and **TargetHeading**.

3. Read the **HTML** property, then call **GetLiveValueFragment** for new headings.

```
TsgcHTMLComponent_Compass *oCompass = new TsgcHTMLComponent_Compass(NULL);
try
{
  oCompass->CompassID = "windDir";
  oCompass->Title = "Wind";
  oCompass->CompassStyle = csRotatingNeedle;
  oCompass->Animate = true;
  oCompass->Heading = 225;
  html = oCompass->HTML;
}
__finally
{
  delete oCompass;
}
```

Heading

Property	Type	Default	Description
Heading	Double	0	The heading in degrees, 0 = north, clockwise.
ShowTarget	Boolean	False	Draws a marker at TargetHeading.
TargetHeading	Double	0	Heading of the target marker in degrees.

Appearance

Property	Type	Default	Description
CompassStyle	TsgcHTMLCompassStyle	csRotatingCard	Rotating card or rotating needle.
ShowCardinals	Boolean	True	Prints N, NE, E … on the card.
ShowDegrees	Boolean	True	Prints the degree scale and the heading readout.
Animate	Boolean	False	Turns the card or needle smoothly, along the shortest path.
Title	string	'' (empty)	Caption drawn below the compass.
CompassID	string	'' (empty)	The id of the root element; falls back to sgcCompass.

Live Updates

Call **GetLiveValueFragment** whenever the reading changes. It stores the new value and returns the whole instrument with `hx-swap-oob="outerHTML"` on its root element. Broadcast that string with the sgcHTMX engine and every browser that shows the page swaps the instrument in place. See Instruments for the full flow.

Output

The inherited **HTML** property returns a `<div>` that carries the id, an inline `<svg>` with `role="img"` and an `aria-label` describing the reading, and an optional small title. No JavaScript library is needed. For a one-off instrument with no component instance, use the static **Build** class function.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) is a control room for a simulated boiler loop that uses the instrumentation components together and keeps them live over the sgcHTMX WebSocket bridge.

Reference

- Properties — 9 properties available on the component.
- Methods — 2 public methods.

- Events — this component has no published events.

TsgcHTMLComponent_Knob

TsgcHTMLComponent_Knob renders a rotary SVG knob that the user turns with the pointer, the keyboard or the mouse wheel, and posts the selected value.

Introduction

TsgcHTMLComponent_Knob draws a rotary control with a scale that sweeps **ArcAngle** degrees, open at the bottom, between **MinValue** and **MaxValue**. The user drags the knob around its center, uses the arrow keys (PageUp and PageDown move ten steps, Home and End jump to the limits) or the mouse wheel. The value snaps to **Step**.

The SVG has `role="slider"` with `aria-valuemin`, `aria-valuemax` and `aria-valuenow`, so it is reachable with the keyboard and announced by screen readers. The **Ranges** collection draws colored bands on the scale, and a range that contains the value also colors the indicator.

Quick Start

1. Drop a **TsgcHTMLComponent_Knob** on the form (or create it at runtime).

2. Set **MinValue**, **MaxValue**, **Step** and **Value**, and a **PostURL** (or **WSSend**) to receive the changes.

3. Read the **HTML** property and serve it; the posted field is **FieldName**.

```
TsgcHTMLComponent_Knob *oKnob = new TsgcHTMLComponent_Knob(NULL);
try
{
  oKnob->KnobID = "setpoint";
  oKnob->Title = "Setpoint";
  oKnob->MinValue = 20;
  oKnob->MaxValue = 90;
  oKnob->Step = 0.5;
  oKnob->Value = 65;
  oKnob->Unit_ = "C";
  oKnob->PostURL = "/api/setpoint";
  html = oKnob->HTML;
}
__finally
{
  delete oKnob;
}
```

Value and Range

Property	Type	Default	Description
Value	Double	0	The value selected by the knob.
MinValue	Double	0	The value at the start of the scale.
MaxValue	Double	100	The value at the end of the scale.

Step	Double	1	Increment the value snaps to; 0 means continuous.
Unit_	string	'' (empty)	Unit text appended to the value readout.

Identity

Property	Type	Default	Description
Title	string	'' (empty)	Caption drawn below the knob.
KnobID	string	'' (empty)	The id of the root element; falls back to sgcKnob.

Appearance

Property	Type	Default	Description
ArcAngle	Integer	270	Sweep of the scale in degrees (60..350), open at the bottom.
Size	Integer	160	Natural width of the knob in pixels.
ShowValue	Boolean	True	Shows the value readout in the center.
KnobColor	string	'' (empty)	CSS color of the value arc and indicator; empty uses the primary color.
Ranges	TsgcHTMLInstrumentRanges	empty collection	Colored bands of the scale (StartValue, EndValue, Color, Caption).

Posting

Property	Type	Default	Description
FieldName	string	'' (empty)	Name of the posted field; empty uses the component id.
PostURL	string	'' (empty)	URL that receives an hx-post each time a change is committed.

PostTarget	string	'' (empty)	hx-target for the response of PostURL; empty discards the response.
WSSend	Boolean	False	Sends the change through the htmx WebSocket extension instead of hx-post.
Disabled	Boolean	False	Renders the control disabled.

Posting the Value

The input components share one posting model. The value lives in a named form element (a hidden input with id `<KnobID>-input`) whose `name` is **FieldName**, or **KnobID** (`sgcKnob` when that is empty too) when FieldName is empty. Because it is a normal named element, the value is posted by a regular form submit when the component sits inside a `<form>`, with no extra code.

For immediate posting set **PostURL**: the element gets `hx-post` with `hx-trigger="sgc-change"`, and the client script fires the `sgc-change` event when a change is committed (when the pointer is released after a drag, or 300 ms after the last key press or wheel step). htmx then posts the element as `FieldName=value`. The response is ignored (`hx-swap="none"`) unless **PostTarget** names a target for it. With **WSSend** the change is sent through the htmx WebSocket extension (`ws-send`) instead of an HTTP request; WSSend wins over PostURL.

Disabled writes the `disabled` attribute and `data-disabled="1"`, so the control ignores the pointer and the keyboard and the browser does not post it. The client script is emitted once per page and uses document level handlers, so it keeps working after the component is replaced by a live update fragment.

Live Updates

When the value changes on the server side (another user, a PLC, an MQTT message), call the GetLive fragment method and broadcast the result with **TsgcHTMX_Engine_Server.BroadcastFragment**: every browser swaps the control in place. **TsgcHTMLInstrumentsMQTTBinder** can do this automatically from MQTT topics. See Instruments for the full flow.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) shows the input components driving a simulated boiler loop over the sgcHTMX WebSocket bridge.

Reference

- Properties — 17 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_ToggleSwitch

TsgcHTMLComponent_ToggleSwitch renders an on/off switch, as a sliding toggle or an industrial rocker, and posts 1 or 0 when it is flipped.

Introduction

TsgcHTMLComponent_ToggleSwitch draws a two state switch. **SwitchStyle** chooses a sliding toggle or a rocker switch, **OnText** and **OffText** label the two positions (the rocker shows I and O when they are empty) and **OnColor** and **OffColor** color the track.

It is built on a real checkbox with `role="switch"`, so it works with the keyboard and screen readers. A hidden input next to it carries the value that is posted: `1` when checked and `0` when not, so the unchecked state is posted too (a plain checkbox posts nothing when unchecked).

Quick Start

1. Drop a **TsgcHTMLComponent_ToggleSwitch** on the form (or create it at runtime).

2. Set **Caption**, **Checked** and a **PostURL** (or **WSSend**).

3. Read the **HTML** property; the server receives `FieldName=1` or `FieldName=0`.

```
TsgcHTMLComponent_ToggleSwitch *oSwitch = new TsgcHTMLComponent_ToggleSwitch(NULL);
try
{
  oSwitch->ToggleSwitchID = "pumpEnable";
  oSwitch->Caption = "Pump";
  oSwitch->SwitchStyle = tssRocker;
  oSwitch->Checked = true;
  oSwitch->PostURL = "/api/pump";
  html = oSwitch->HTML;
}
__finally
{
  delete oSwitch;
}
```

State

Property	Type	Default	Description
Checked	Boolean	False	The state of the switch.

Identity

Property	Type	Default	Description
Caption	string	'' (empty)	Label shown next to the switch.
ToggleSwitchID	string	'' (empty)	The id of the root element; falls back to sgcToggleSwitch.

Appearance

Property	Type	Default	Description
Switch-Style	TsgcHTMLToggleSwitch-Style	tssTog-gle	Sliding toggle or rocker switch.
OnText	string	'' (empty)	Text of the on position; empty shows no text (I on a rocker).
OffText	string	'' (empty)	Text of the off position; empty shows no text (O on a rocker).
OnColor	string	'' (empty)	CSS color of the on state; empty uses the suc-cess green.
OffColor	string	'' (empty)	CSS color of the off state; empty uses grey.
Size	Integer	24	Height of the switch in pixels.

Posting

Property	Type	Default	Description
FieldName	string	'' (empty)	Name of the posted field; empty uses the component id.
PostURL	string	'' (empty)	URL that receives an hx-post each time a change is committed.
PostTarget	string	'' (empty)	hx-target for the response of PostURL; empty discards the re-sponse.
WSSend	Boolean	False	Sends the change through the htmx WebSocket extension in-stead of hx-post.

Disabled	Boolean	False	Renders the control disabled.

Posting the Value

The input components share one posting model. The value lives in a named form element (a hidden input holding `1` or `0`) whose `name` is **FieldName**, or **ToggleSwitchID** (`sgcToggleSwitch` when that is empty too) when FieldName is empty. Because it is a normal named element, the value is posted by a regular form submit when the component sits inside a `<form>`, with no extra code.

For immediate posting set **PostURL**: the element gets `hx-post` with `hx-trigger="sgc-change"`, and the client script fires the `sgc-change` event when a change is committed (each time the switch is flipped). htmx then posts the element as `FieldName=value`. The response is ignored (`hx-swap="none"`) unless **PostTarget** names a target for it. With **WSSend** the change is sent through the htmx WebSocket extension (`ws-send`) instead of an HTTP request; WSSend wins over PostURL.

Disabled writes the `disabled` attribute and `data-disabled="1"`, so the control ignores the pointer and the keyboard and the browser does not post it. The client script is emitted once per page and uses document level handlers, so it keeps working after the component is replaced by a live update fragment.

Live Updates

When the value changes on the server side (another user, a PLC, an MQTT message), call the GetLive fragment method and broadcast the result with **TsgcHTMX_Engine_Server.BroadcastFragment**: every browser swaps the control in place. **TsgcHTMLInstrumentsMQTTBinder** can do this automatically from MQTT topics. See Instruments for the full flow.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) shows the input components driving a simulated boiler loop over the sgcHTMX WebSocket bridge.

Reference

- Properties — 14 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_PushButton

TsgcHTMLComponent_PushButton renders a command button with a status lamp, as a click button or a momentary (hold to run) button, with an optional in-page confirmation.

Introduction

TsgcHTMLComponent_PushButton draws a Bootstrap button (**ButtonStyle**) with a small LED lamp. The lamp shows **LEDState** in **LEDColor**, so the server can reflect the real state of the machine the button commands (for example "running") with **GetLiveLEDFragment**.

A normal button posts `1` on each click. With **Momentary** it posts `1` when pressed and `0` when released (pointer or Space and Enter keys) and exposes `aria-pressed`, like a jog or hold to run button. With **Confirm** a click first shows the confirmation text with a confirm and a cancel button inside the page; their captions come from the locale (**ConfirmText** and **CancelText** of the locale (sgcHTMLDefaultLocale)). Confirm is ignored when Momentary is True.

Quick Start

1. Drop a **TsgcHTMLComponent_PushButton** on the form (or create it at runtime).

2. Set **Caption**, **ButtonStyle**, an optional **Confirm** text and a **PostURL** (or **WSSend**).

3. Read the **HTML** property; call **GetLiveLEDFragment** when the commanded state changes.

```
TsgcHTMLComponent_PushButton *oButton = new TsgcHTMLComponent_PushButton(NULL);
try
{
  oButton->PushButtonID = "estop";
  oButton->Caption = "Emergency stop";
  oButton->ButtonStyle = bsDanger;
  oButton->Confirm = "Stop the whole line?";
  oButton->PostURL = "/api/estop";
  html = oButton->HTML;
}
__finally
{
  delete oButton;
}
```

Behavior

Property	Type	Default	Description
Momentary	Boolean	False	Posts 1 while pressed and 0 on release instead of 1 per click.
Confirm	string	'' (empty)	Confirmation text shown in the page before the click is posted.

Identity

Property	Type	Default	Description
Caption	string	'' (empty)	Text of the button.
PushButtonID	string	'' (empty)	The id of the root element; falls back to sgcPushButton.

Appearance

Property	Type	Default	Description
ButtonStyle	TsgcHTMLButtonStyle	bsPrimary	Bootstrap style of the button.
LEDState	Boolean	False	Lights the status lamp of the button.
LEDColor	string	'' (empty)	CSS color of the lit lamp; empty uses the success green.

Posting

Property	Type	Default	Description
FieldName	string	'' (empty)	Name of the posted field; empty uses the component id.
PostURL	string	'' (empty)	URL that receives an hx-post each time a change is committed.
PostTarget	string	'' (empty)	hx-target for the response of PostURL; empty discards the response.
WSSend	Boolean	False	Sends the change through the htmx WebSocket extension instead of hx-post.
Disabled	Boolean	False	Renders the control disabled.

Posting the Value

The input components share one posting model. The value lives in a named form element (a hidden input) whose `name` is **FieldName**, or **PushButtonID** (`sgcPushButton` when that is empty too) when FieldName is empty. Because it

is a normal named element, the value is posted by a regular form submit when the component sits inside a `<form>`, with no extra code.

For immediate posting set **PostURL**: the element gets `hx-post` with `hx-trigger="sgc-change"`, and the client script fires the `sgc-change` event when a change is committed (on each click, after the in-page confirmation when Confirm is set, or on press and on release when Momentary is True). htmx then posts the element as `FieldName=value`. The response is ignored (`hx-swap="none"`) unless **PostTarget** names a target for it. With **WSSend** the change is sent through the htmx WebSocket extension (`ws-send`) instead of an HTTP request; WSSend wins over PostURL.

Disabled writes the `disabled` attribute and `data-disabled="1"`, so the control ignores the pointer and the keyboard and the browser does not post it. The client script is emitted once per page and uses document level handlers, so it keeps working after the component is replaced by a live update fragment.

Live Updates

When the value changes on the server side (another user, a PLC, an MQTT message), call the GetLive fragment method and broadcast the result with **TsgcHTMX_Engine_Server.BroadcastFragment**: every browser swaps the control in place. **TsgcHTMLInstrumentsMQTTBinder** can do this automatically from MQTT topics. See Instruments for the full flow.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) shows the input components driving a simulated boiler loop over the sgcHTMX WebSocket bridge.

Reference

- Properties — 12 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_NumericStepper

TsgcHTMLComponent_NumericStepper renders a number box with decrease and increase buttons that repeat while held, clamps the value to its range and posts it.

Introduction

TsgcHTMLComponent_NumericStepper draws a `<input type="number">` between a minus and a plus button. Holding a button repeats the step (after 400 ms, then every 80 ms), the arrow keys and PageUp and PageDown also change the value, and the result is always clamped to **MinValue**..**MaxValue** and shown with **Decimals** decimals.

The accessible labels of the two buttons come from the locale: **DecreaseText** and **IncreaseText** of the locale (sgcHTMLDefaultLocale) (default "Decrease" and "Increase"), so translate them there.

Quick Start

1. Drop a **TsgcHTMLComponent_NumericStepper** on the form (or create it at runtime).

2. Set **MinValue**, **MaxValue**, **Step**, **Value** and a **PostURL** (or **WSSend**).

3. Read the **HTML** property and serve it.

```
TsgcHTMLComponent_NumericStepper *oStepper = new TsgcHTMLComponent_NumericStepper(NULL);
try
{
  oStepper->NumericStepperID = "batchSize";
  oStepper->MinValue = 1;
  oStepper->MaxValue = 500;
  oStepper->Step = 5;
  oStepper->Value = 100;
  oStepper->Unit_ = "pcs";
  oStepper->PostURL = "/api/batch";
  html = oStepper->HTML;
}
__finally
{
  delete oStepper;
}
```

Value and Range

Property	Type	Default	Description
Value	Double	0	The current value.
MinValue	Double	0	Lowest value accepted.
MaxValue	Double	100	Highest value accepted.

Step	Double	1	Amount added or subtracted by one button press.
Decimals	Integer	-1	Decimals shown; -1 derives them from Step.
Unit_	string	'' (empty)	Unit text shown after the number box.

Identity

Property	Type	Default	Description
NumericStepperID	string	'' (empty)	The id of the root element; falls back to sgcNumeric-Stepper.

Posting

Property	Type	Default	Description
FieldName	string	'' (empty)	Name of the posted field; empty uses the component id.
PostURL	string	'' (empty)	URL that receives an hx-post each time a change is committed.
PostTarget	string	'' (empty)	hx-target for the response of PostURL; empty discards the re-sponse.
WSSend	Boolean	False	Sends the change through the htmx WebSocket extension in-stead of hx-post.
Disabled	Boolean	False	Renders the control disabled.

Posting the Value

The input components share one posting model. The value lives in a named form element (the number input itself) whose `name` is **FieldName**, or **NumericStepperID** (`sgcNumericStepper` when that is empty too) when FieldName is empty. Because it is a normal named element, the value is posted by a regular form submit when the component sits inside a `<form>`, with no extra code.

For immediate posting set **PostURL**: the element gets `hx-post` with `hx-trigger="sgc-change"`, and the client script fires the `sgc-change` event when a change is committed (300 ms after the last button press, key press or edit). htmx then posts the element as `FieldName=value`. The response is ignored (`hx-swap="none"`) unless **PostTarget** names a target for it. With **WSSend** the change is sent through the htmx WebSocket extension (`ws-send`) instead of an HTTP request; WSSend wins over PostURL.

Disabled writes the `disabled` attribute and `data-disabled="1"`, so the control ignores the pointer and the keyboard and the browser does not post it. The client script is emitted once per page and uses document level handlers, so it keeps working after the component is replaced by a live update fragment.

Live Updates

When the value changes on the server side (another user, a PLC, an MQTT message), call the GetLive fragment method and broadcast the result with **TsgcHTMX_Engine_Server.BroadcastFragment**: every browser swaps the control in place. **TsgcHTMLInstrumentsMQTTBinder** can do this automatically from MQTT topics. See Instruments for the full flow.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) shows the input components driving a simulated boiler loop over the sgcHTMX WebSocket bridge.

Reference

- Properties — 12 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_StripChart

TsgcHTMLComponent_StripChart renders a scrolling multi pen strip chart on a canvas, like a chart recorder, fed live with samples pushed from the server.

Introduction

TsgcHTMLComponent_StripChart plots one trace per pen of the **Pens** collection against time. The newest sample is at the right edge and the chart scrolls left, showing the last **TimeWindow** seconds. Every pen keeps a ring buffer of **MaxSamples** samples in the browser.

The component emits a `<canvas>` and a small inline runtime (emitted once per page, no .js file and no resources), and registers a JavaScript object for the instance in `window.sgcInstruments[<id>]`. The server never renders the trace again: it sends samples with **GetLivePushScript**, which returns an out-of-band fragment that appends a self-removing `<script>` to the instrument, and the script calls `push` on the client object. NaN and infinite values are sent as `null` and leave a gap in the trace.

The Y axis spans **MinValue**..**MaxValue** (there is no auto ranging) and is labelled with "nice" ticks (1, 2, 2.5 or 5 times a power of ten) whose count follows the plot height, with gridlines at the ticks. A pen can have its own scale with its own MinValue and MaxValue. The legend shows each pen with its last value, and a pause button freezes the display while samples keep arriving.

Quick Start

1. Drop a **TsgcHTMLComponent_StripChart** on the form (or create it at runtime).

2. Add one pen per signal to **Pens** and set **MinValue**, **MaxValue** and **TimeWindow**.

3. Serve the **HTML** property, then broadcast **GetLivePushScript** with one value per pen for every sample.

```
TsgcHTMLComponent_StripChart *oStrip = new TsgcHTMLComponent_StripChart(NULL);
try
{
  oStrip->StripChartID = "boilerTrend";
  oStrip->Title = "Boiler";
  oStrip->MinValue = 0;
  oStrip->MaxValue = 120;
  oStrip->TimeWindow = 120;
  oStrip->Pens->Add()->Caption = "Temperature";
  oStrip->Pens->Add()->Caption = "Pressure";
  html = oStrip->HTML;
  double vValues[] = {78.4, 2.1};
  HTMXServer->BroadcastFragment(oStrip->GetLivePushScript(vValues, 1));
}
__finally
{
  delete oStrip;
}
```

Pens and Scale

Property	Type	Default	Description
Pens	TsgcHTML-StripChartPens	empty col-lection	The traces of the chart (Caption, Color, LineWidth, MinValue, MaxValue, Visible).
MinValue	Double	0	Bottom of the Y scale.
MaxValue	Double	100	Top of the Y scale.
TimeWin-dow	Integer	60	Visible time span in seconds.
MaxSam-ples	Integer	1000	Size of the ring buffer of every pen in the browser.

Identity

Property	Type	Default	Description
Title	string	'' (empty)	Caption drawn above the chart.
StripChartID	string	'' (empty)	The id of the root element; falls back to sgcStripChart.

Appearance

Property	Type	Default	Description
GridLines	Boolean	True	Draws the horizontal and vertical gridlines.
ShowLegend	Boolean	True	Shows the legend with the pen captions.
ShowValues	Boolean	True	Shows the last value of each pen in the legend.
ShowPauseButton	Boolean	True	Shows the client side pause / resume button.
PauseText	string	'Pause'	Caption of the pause button while the chart runs.
ResumeText	string	'Resume'	Caption of the pause button while the chart is paused.
BackgroundColor	string	'' (empty)	CSS color of the plot area; empty is transparent.
GridColor	string	'' (empty)	CSS color of the gridlines; empty uses a translucent grey.
DrawingWidth	Integer	600	Width of the chart in pixels.

DrawingHeight	Integer	200	Height of the chart in pixels.

Client API

`window.sgcInstruments[<StripChartID>]` exposes the chart to page scripts:

Member	Description
`push(samples[, tsMs])`	Appends one sample per pen (an array, or a single number for one pen), optionally with a Unix time in milliseconds.
`pause()`, `resume()`, `toggle()`	Freezes, resumes or toggles the display; samples are still stored while paused.
`isPaused()`	True while the display is paused.
`size(i)`	Number of samples stored for pen `i`.
`values(i)`	Array with the stored values of pen `i`, oldest first.
`ticks()`	The Y axis ticks in use.
`draw()`	Redraws the canvas.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) shows a strip chart and an oscilloscope fed live over the sgcHTMX WebSocket bridge. **TsgcHTMLInstrumentsMQTTBinder** with the `ibtPush` target feeds a strip chart straight from MQTT.

Reference

- Properties — 17 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Oscilloscope

TsgcHTMLComponent_Oscilloscope renders an oscilloscope screen on a canvas with a division grid, per channel scale, edge trigger and optional phosphor persistence, fed with whole frames from the server.

Introduction

TsgcHTMLComponent_Oscilloscope draws a scope screen of **DivisionsX** by **DivisionsY** divisions. Each item of **Channels** is a trace with its own volts per division and offset. The server pushes a whole frame (an array of samples) per channel, and the frame replaces the previous one of that channel.

The component emits a `<canvas>` and a small inline runtime (emitted once per page, no .js file and no resources), and registers a JavaScript object for the instance in `window.sgcInstruments[<id>]`. The server never renders the trace again: it sends samples with **GetLivePushScript**, which returns an out-of-band fragment that appends a self-removing `<script>` to the instrument, and the script calls `push` on the client object. NaN and infinite values are sent as `null` and leave a gap in the trace.

The trigger aligns the frames of **TriggerChannel**: the trace starts at the first rising crossing of **TriggerLevel**. With **Persistence** old frames fade out like phosphor instead of being cleared. The screen shows the time per division, and the client object computes per channel statistics (peak to peak, average, minimum and maximum).

Quick Start

1. Drop a **TsgcHTMLComponent_Oscilloscope** on the form (or create it at runtime).

2. Add the **Channels**, set **VoltsPerDiv** on each and the trigger properties.

3. Serve the **HTML** property, then broadcast **GetLivePushScript** with a frame of samples per channel.

```
TsgcHTMLComponent_Oscilloscope *oScope = new TsgcHTMLComponent_Oscilloscope(NULL);
try
{
  oScope->OscilloscopeID = "vibration";
  oScope->Title = "Vibration";
  oScope->TimePerDiv = 0.002;
  oScope->TriggerLevel = 0.5;
  TsgcHTMLOscilloscopeChannel *oChannel = oScope->Channels->Add();
  oChannel->Caption = "CH1";
  oChannel->VoltsPerDiv = 0.5;
  html = oScope->HTML;
}
__finally
{
  delete oScope;
}
```

Channels and Trigger

Property	Type	Default	Description

Channels	TsgcHTMLOscillo-scopeChannels	empty col-lection	The traces of the scope (Caption, Color, VoltsPerDiv, Offset, Visible).
DivisionsX	Integer	10	Number of horizontal divisions of the grid.
DivisionsY	Integer	8	Number of vertical divisions of the grid.
TimePer-Div	Double	0.001	Seconds per horizontal division, shown as a label.
Trigger-Mode	TsgcHTMLOscillo-scopeTriggerMode	otmAuto	How frames are aligned to the trigger.
Trigger-Level	Double	0	Level of the rising edge trigger.
Trigger-Channel	Integer	0	Index of the channel that triggers.

Identity

Property	Type	Default	Description
Title	string	'' (empty)	Caption of the scope.
OscilloscopeID	string	'' (empty)	The id of the root element; falls back to sgcOscilloscope.

Appearance

Property	Type	Default	Description
Persistence	Boolean	False	Fades old frames out like phosphor instead of clearing them.
GridColor	string	'' (empty)	CSS color of the grid; empty uses a translucent white.
TraceBack-ground	string	'#101418'	CSS color of the screen.
DrawingWidth	Integer	500	Width of the scope screen in pixels.
DrawingHeight	Integer	400	Height of the scope screen in pixels.

Client API

`window.sgcInstruments[<OscilloscopeID>]` exposes the scope to page scripts:

Member	Description
`push(ch, frame)`	Replaces the frame of channel `ch`. Returns false when the channel does not exist or, in otmNormal mode, when the trigger channel frame has no trigger crossing.
`trigger()`	Index of the sample where the last trigger fired.
`stats(ch)`	Statistics of the last frame of the channel: `{vpp, vavg, min, max}` (peak to peak, average, minimum, maximum), or null.
`frame(ch)`	The last frame of the channel, or null.
`draw()`	Redraws the canvas.

Demo

Demos\60.HTML\01.RunTime\20.Instruments (and its form based twin in **02.DesignTime\20.Instruments**) shows a strip chart and an oscilloscope fed live over the sgcHTMX WebSocket bridge.

Reference

- Properties — 14 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_SCADAPanel

TsgcHTMLComponent_SCADAPanel renders a process mimic diagram (tanks, valves, pumps, motors, fans, heat exchangers, sensors and labels joined by pipes) as SVG, with live state, flow animation and clickable symbols.

Introduction

TsgcHTMLComponent_SCADAPanel draws a synoptic panel of **DrawingWidth** by **DrawingHeight** SVG user units that scales down with its container. The **Symbols** collection places the equipment and the **Pipes** collection draws the lines between them. Every symbol is colored by its **State**, a tank shows its **Level**, a sensor its **Value**, and a pipe can animate its flow.

Each symbol and each pipe is rendered as its own nested `<svg>` with the id `<SCADAPanelID>-<SymbolID>` or `<SCADAPanelID>-<PipeID>` and `overflow="visible"`. A nested svg (not a bare `<g>`) is what lets htmx swap a single symbol out of band and keep it as SVG, so **GetLiveSymbolFragment** and **GetLivePipeFragment** update one element instead of the whole panel. An item with an empty id uses `sym<Index>` or `pipe<Index>`; symbols and pipes share one id namespace, so do not give a symbol and a pipe the same id.

Quick Start

1. Drop a **TsgcHTMLComponent_SCADAPanel** on the form (or create it at runtime).

2. Add the equipment to **Symbols** and the lines to **Pipes**, each with an id.

3. Serve the **HTML** property, and when a state changes broadcast **GetLiveSymbolFragment** or **GetLivePipeFragment**.

```
TsgcHTMLComponent_SCADAPanel *oPanel = new TsgcHTMLComponent_SCADAPanel(NULL);
try
{
  oPanel->SCADAPanelID = "plant";
  TsgcHTMLSCADASymbol *oTank = oPanel->Symbols->Add();
  oTank->SymbolID = "tank1";
  oTank->Kind = sskTank;
  oTank->X = 40;
  oTank->Y = 60;
  oTank->Level = 72;
  TsgcHTMLSCADAPipe *oPipe = oPanel->Pipes->Add();
  oPipe->PipeID = "line1";
  oPipe->Points = "120,175 240,175";
  oPipe->Flow = true;
  html = oPanel->HTML;
}
__finally
{
  delete oPanel;
}
```

Content

Property	Type	Default	Description

Symbols	TsgcHTMLSCADASymbols	empty collection	The equipment symbols of the panel.
Pipes	TsgcHTMLSCADAPipes	empty collection	The pipes drawn between the symbols.

Identity

Property	Type	Default	Description
Title	string	'' (empty)	Caption of the panel.
SCADAPanelID	string	'' (empty)	The id of the root element; falls back to sgcSCADAPanel.

Appearance

Property	Type	Default	Description
BackgroundColor	string	'' (empty)	CSS color of the panel; empty uses a light grey.
ShowGrid	Boolean	False	Draws a 20 unit layout grid.
DrawingWidth	Integer	800	Width of the panel (SVG user units) in pixels.
DrawingHeight	Integer	450	Height of the panel (SVG user units) in pixels.

Symbol Kinds

Kind	Default size	Drawing
sskTank	80 x 120	Vertical tank whose liquid fill follows `Level` (0..100 %), with the level printed.
sskValve	40 x 28	Bow tie valve; uses the states ssOpen, ssClosed and ssTransit.
sskPump	50 x 50	Centrifugal pump; the impeller spins while the state is ssOn.
sskMotor	50 x 40	Motor with an M mark; colored by ssOn and ssOff.
sskFan	50 x 50	Fan whose blades spin while the state is ssOn.

sskHeatEx-changer	60 x 60	Heat exchanger symbol.
sskSensor	40 x 40	Round instrument bubble that shows `Value` with `Decimals` and `Unit_` .
sskLabel	120 x 24	Text label (Caption); it shows `Value` as well only when `Unit_` is not empty.

States

Every symbol gets the class `sgc-scada-<state>` and the attributes `data-state` and `data-kind`, so you can restyle it with CSS. Animations (spinning, blinking, flow) are switched off when the browser asks for reduced motion.

State	Color	Meaning
ssOff	#adb5bd (grey)	Stopped equipment.
ssOn	#2f9e44 (green)	Running equipment; pumps and fans spin.
ssFault	#e03131 (red)	Fault; the symbol blinks (class `sgc-scada-alarm`), except a label.
ssOpen	#2f9e44 (green)	Open valve.
ssClosed	#adb5bd (grey)	Closed valve.
ssTransit	#f59f00 (amber)	Valve moving between open and closed.

Symbol Properties

Property	De-fault	Description
SymbolID	empty	Id of the symbol, used by FindSymbol, GetLiveSymbolFragment and ClickURL posts.
Kind	ssk-Tank	The symbol drawn (see the kinds table).
X, Y	0	Top left corner of the symbol box, in SVG user units of the panel.
Width, Height	0	Size of the box; 0 uses the default size of the kind.
Rotation	0	0, 90, 180 or 270 degrees clockwise around the center of the box.

Caption	empty	Name drawn with the symbol and used as its accessible label.
State	ssOff	Operating state (see the states table).
Level	0	Tank fill in percent, 0..100.
Value	0	Reading of a sensor, or of a label with Unit_.
Unit_	empty	Unit printed after Value.
Decimals	1	Maximum decimals of Value and Level (0..6). Trailing zeros are trimmed, so 12.50 with Decimals 2 prints 12.5.
ClickURL	empty	When set, the symbol is a button that posts to this URL (see Commands).

Pipe Properties

Property	Default	Description
PipeID	empty	Id of the pipe, used by FindPipe and GetLivePipeFragment.
Points	empty	Polyline of the pipe: `"x1,y1 x2,y2 ..."` (see Points syntax).
Width	6	Stroke width of the pipe.
Color	empty	CSS color of the pipe; empty uses #868e96.
Flow	False	Animates dashes along the pipe to show flow (static when the browser asks for reduced motion).
FlowDirection	sfdForward	sfdForward runs from the first point to the last, sfdReverse the other way.
FlowColor	empty	CSS color of the flow dashes; empty uses #a5d8ff.

Points Syntax

Points is a list of `x,y` pairs in SVG user units, for example `"20,100 140,100 140,40"`. Numbers use a dot as decimal separator and may have a sign. Pairs and coordinates can be separated by spaces, commas, tabs or line breaks.

The value is sanitized before it reaches the page: any other character (letters, quotes, brackets), a malformed or out of range number, fewer than two points or an odd count of numbers rejects the whole pipe, which is then rendered as an empty wrapper. The numbers are written back with at most two decimals.

Commands (ClickURL)

A symbol with **ClickURL** becomes a button (`role="button"`, `tabindex="0"`): a click, Enter or Space sends an `hx-post` to the URL with the values `{"panel": "<SCADAPanelID>", "symbol": "<SymbolID>"}` and discards the response (`hx-swap="none"`). The URL is sanitized, so a `javascript:` URL becomes `#`.

In the handler read `panel` and `symbol`, act on the equipment, change the symbol and broadcast **GetLiveSymbol-Fragment**, so every operator sees the new state.

Demo

Demos\60.HTML\01.RunTime\21.SCADA_IoT (and its form based twin in **02.DesignTime\21.SCADA_IoT**) shows a process plant mimic updated live from MQTT through **TsgcHTMLInstrumentsMQTTBinder**, with clickable pumps and valves.

Reference

- Properties — 8 properties available on the component.
- Methods — 5 public methods.
- Events — this component has no published events.

TsgcHTMLInstrumentsBinder

TsgcHTMLInstrumentsBinder connects topics to sgcHTML instruments without any transport: every message you pass to ProcessMessage updates the bound instruments and broadcasts their live fragments to the browsers.

Introduction

TsgcHTMLInstrumentsBinder is a non visual component that holds the **Bindings** between topics and instruments and a **TsgcHTMX_Engine_Server** (**HTMXServer**) that broadcasts the fragments. It does not receive messages by itself: you call **ProcessMessage(Topic, Payload)** from any source, a custom MQTT client, an AMQP or Kafka consumer, a WebSocket, a serial port or a simulator. Each enabled binding whose topic filter matches extracts the value (optionally with a JSONPath), scales it, calls the live fragment method of the instrument and broadcasts the fragment, so every browser that shows the instrument updates at once.

TsgcHTMLInstrumentsMQTTBinder descends from this class and adds only an MQTT client that feeds ProcessMessage. Everything described here (bindings, targets, wildcards, JSONPath, throttling, events) applies to both.

Quick Start

1. Drop a **TsgcHTMX_Engine_Server**, the instruments and a **TsgcHTMLInstrumentsBinder** on the form.

2. Assign **HTMXServer** and add one item to **Bindings** per topic and instrument.

3. Call **ProcessMessage** for every message your source delivers. If a binding uses **MinIntervalMs**, call **ApplyPending** from a timer.

```
oBinder->HTMXServer = HTMXServer;
TsgcHTMLInstrumentBinding *oBinding = oBinder->Bindings->Add();
oBinding->Topic = "plant/boiler/temperature";
oBinding->Instrument = oGauge;
oBinding->Target = ibtValue;
oBinder->ProcessMessage("plant/boiler/temperature", "72.5");
```

Feeding the Binder from Any Source

The topic you pass is matched with the MQTT rules (levels separated by `/`), whatever the source is. Build a topic from the routing information of your source and pass the message text as the payload.

AMQP. Call ProcessMessage from OnAMQPBasicDeliver of a **TsgcWSPClient_AMQP**. AMQP routing keys use dots, so convert them to slashes and write the binding topics with slashes (`plant/+/temperature`).

Kafka, WebSocket or a custom client. In the message event of the client, pass a topic (the Kafka topic name, a channel, a device id) and the text of the message: `oBinder.ProcessMessage(vTopic, vText)`.

A simulator. A timer or a worker thread can generate the values; ProcessMessage is thread safe.

ASP.NET Core

TsgcHTMLInstrumentsBinder is compiled in the full .NET build and in the standalone ASP.NET Core build of sgcHTML (the MQTT binder is not available in the standalone build). In ASP.NET Core the pages are served by the sgcHTML middleware and there is no engine WebSocket server to assign to HTMXServer, so handle the .NET only OnBroadcast event and send each fragment with `ISgcHtmlHub.BroadcastAsync`. When OnBroadcast has a handler, HTMXServer is not used.

Bindings, Targets, JSONPath and Values

The binding properties, the **Target** values and the instruments each target supports, the topic wildcards, the JSON-Path syntax and the value conversions are described in TsgcHTMLInstrumentsMQTTBinder and in Bindings; they are implemented in this class and work the same for any source.

Throttling

With **MinIntervalMs** greater than 0 a binding applies at most one message per interval. Messages that arrive too early are kept as the pending value; call **ApplyPending** from a timer so the last value is shown when the source goes quiet.

Thread Safety

The instruments are not thread safe, so ProcessMessage and ApplyPending run the whole update, render and broadcast inside a critical section. They can be called from any thread. OnBeforeUpdate, OnAfterUpdate, On-BindingError (and OnBroadcast in .NET) are called inside that section, from the calling thread: do not touch VCL controls there without synchronizing, and keep the handlers short.

Extending

The processing steps are protected virtual methods that a descendant can override: **DoProcessMessage** (match the bindings), **DoProcessBinding** (throttle and apply one binding), **DoExtractJSONValue** (read the JSONPath), **DoGetFragment** (update the instrument and render its fragment), **DoBroadcast** (send the fragment) and **DoBindingError** (raise OnBindingError). For example, override DoBroadcast to send the fragments through your own server.

Requirements

The binder is available when sgcHTML and sgcHTMX are enabled (`SGC_HTML` and `SGC_HTMX`). Unit: `sgcHTML_Instruments_MQTT`.

Reference

- Properties — Bindings and HTMXServer.
- Methods — ProcessMessage, ApplyPending and TopicMatches.
- Events — OnBeforeUpdate, OnAfterUpdate, OnBindingError and OnBroadcast (.NET only).

TsgcHTMLInstrumentsMQTTBinder

TsgcHTMLInstrumentsMQTTBinder connects MQTT topics to sgcHTML instruments with no code: every message received on a bound topic updates the instrument and broadcasts its live fragment to the browsers.

Introduction

TsgcHTMLInstrumentsMQTTBinder is a non visual component that sits between a **TsgcWSPClient_MQTT** (**MQTTClient**) and a **TsgcHTMX_Engine_Server** (**HTMXServer**). Each item of **Bindings** maps a topic filter to an instrument and a target. When a message arrives on a matching topic the binder extracts the value (optionally with a JSONPath), scales it, calls the live fragment method of the instrument and broadcasts the fragment with **HTMXServer.BroadcastFragment**, so every browser that shows the instrument updates at once.

TsgcHTMLInstrumentsMQTTBinder descends from TsgcHTMLInstrumentsBinder, the transport free binder, and adds only **MQTTClient** and **SubscribeAll**: every message published to the client is passed to the inherited ProcessMessage. Bindings, HTMXServer, ApplyPending, TopicMatches and the events are inherited. To feed the instruments from another source (AMQP, Kafka, a WebSocket, a simulator) use TsgcHTMLInstrumentsBinder directly.

Call **SubscribeAll** once the MQTT client is connected to subscribe to every bound topic. The events let you rewrite or veto a value (**OnBeforeUpdate**), observe what was sent (**OnAfterUpdate**) and log bad payloads (**OnBindingError**).

Quick Start

1. Drop a **TsgcWSPClient_MQTT**, a **TsgcHTMX_Engine_Server**, the instruments and a **TsgcHTMLInstrumentsMQTTBinder** on the form.

2. Assign **MQTTClient** and **HTMXServer**, and add one item to **Bindings** per topic and instrument.

3. After the MQTT client connects, call **SubscribeAll**. If a binding uses **MinIntervalMs**, call **ApplyPending** from a timer.

```
oBinder->MQTTClient = MQTTClient;
oBinder->HTMXServer = HTMXServer;
TsgcHTMLInstrumentBinding *oBinding = oBinder->Bindings->Add();
oBinding->Topic = "plant/boiler/temperature";
oBinding->Instrument = oGauge;
oBinding->Target = ibtValue;
oBinding = oBinder->Bindings->Add();
oBinding->Topic = "plant/+/status";
oBinding->Instrument = oPanel;
oBinding->Target = ibtSymbolState;
oBinding->ItemID = "pump1";
oBinding->JSONPath = "pump.state";
oBinder->SubscribeAll();
```

Connections

Property	Type	Default	Description
MQTTClient	TsgcWSPClient_MQTT	nil	The MQTT client whose messages feed the bindings.
HTMXServ-er	TsgcHTMX_Engine_Server	nil	The sgcHTMX server that broadcasts the frag-ments.

Bindings

Property	Type	Default	Description
Bindings	TsgcHTMLInstrumentBindings	empty collection	The topic to instrument mappings.

Binding Properties

Property	De-fault	Description
Topic	empty	MQTT topic filter; wildcards `+` and `#` are allowed.
Instru-ment	nil	The sgcHTML instrument that receives the value (any TsgcHTMLComponent; the Target decides which ones work).
Target	ibtVal-ue	Which live API of the instrument is fed (see the Targets table).
ItemID	empty	Symbol id, pipe id or tile id for the SCADA and tile targets; channel index for an oscilloscope.
JSONPath	empty	Dotted path of the value inside a JSON payload; empty uses the whole payload.
Scale	1	Multiplier applied to numeric values.
Offset	0	Added after Scale: value * Scale + Offset.
OnValue	'ON'	Extra text accepted as True by the boolean targets.
OffValue	'OFF'	Extra text accepted as False by the boolean targets.
MinInter-valMs	0	Throttle: at most one applied message per interval; 0 applies every message.
Enabled	True	False ignores the binding.

Targets

Target selects the live API that receives the value. Each target supports only some instruments; any other class raises **OnBindingError**.

Target	Value expected	Instruments	Update
ibtValue	Number	Gauge, LinearGauge, LEDBar, Knob, NumericStepper, Slider (rounded to Integer), Odometer, VUMeter, Compass	Their GetLiveValueFragment.
ibtLevel	Number	LinearGauge, LEDBar	GetLiveValueFragment (the level of the bar or tank).
ibtState	Boolean	LED, ToggleSwitch, PushButton (its lamp)	GetLiveStateFragment, or GetLiveLEDFragment for the push button.
ibtText	Text	SegmentDisplay	GetLiveTextFragment. A numeric payload is scaled only when Scale or Offset differ from 1 and 0.
ibtSymbolState	SCADA state	SCADAPanel (ItemID = SymbolID)	Sets State of the symbol, then GetLiveSymbolFragment.
ibtSymbolLevel	Number	SCADAPanel (ItemID = SymbolID)	Sets Level of the symbol, then GetLiveSymbolFragment.
ibtSymbolValue	Number	SCADAPanel (ItemID = SymbolID)	Sets Value of the symbol, then GetLiveSymbolFragment.
ibtPipeFlow	Boolean	SCADAPanel (ItemID = PipeID)	Sets Flow of the pipe, then GetLivePipeFragment.
ibtTile	Annunciator state	Annunciator (ItemID = tile id)	Sets the State of the tile; when it changes, the tile is marked not acknowledged. Then GetLiveTileFragment.
ibtPush	Number or array	StripChart, Oscilloscope (ItemID = channel index)	GetLivePushScript. A JSON array `[a,b,...]` gives one value per pen (strip chart) or a frame (oscilloscope); every number is scaled.

Topic Wildcards

Topic follows the MQTT filter rules: `+` matches one level (`plant/+/temp` matches `plant/boiler/temp`), `#` matches all remaining levels and must be last (`plant/#`), and topics that start with `$` are not matched by a leading wildcard. Every enabled binding whose filter matches the topic of a message is processed, in collection order. The same rules are available as the class function **TopicMatches**.

JSONPath

When the payload is JSON, **JSONPath** picks the value: a dotted list of member names, where a numeric level is an array index. For `{"boiler":{"temps":[71.5,72.0]}}` the path `boiler.temps.0` gives `71.5`, and for a top level array `[5,6]` the path `0` gives `5`. A path that ends at an array returns it as `[a,b,...]` (useful for ibtPush), a JSON null gives an empty value, and a path that ends at an object, a missing member or an invalid payload fires **OnBindingError**. With an empty JSONPath the whole payload is the value.

Values

Numeric targets apply `value * Scale + Offset`, which converts units (for example Scale 0.1 for tenths of a degree).

Boolean targets (ibtState, ibtPipeFlow) accept `true`, `1`, `on` and **OnValue** as True, and `false`, `0`, `off` and **OffValue** as False, case insensitive.

ibtSymbolState accepts a state name (`off`, `on`, `fault`, `open`, `closed`, `transit`), its ordinal (0..5) or a boolean (on / off). ibtTile accepts `normal`, `advisory`, `warning`, `alarm`, the ordinal (0..3) or a boolean (True is alarm, False is normal).

Throttling

With **MinIntervalMs** greater than 0 a binding applies at most one message per interval, which protects the browsers from a sensor that publishes hundreds of messages a second. Messages that arrive too early are kept as the pending value; call **ApplyPending** from a timer so the last value is shown when the publisher goes quiet.

Handler Chaining

The MQTT client has a single OnMQTTPublish event. The binder chains it: when **MQTTClient** is assigned it keeps the current handler and calls it first on every message, then passes the topic and the text to **ProcessMessage**. Assign your handler before MQTTClient; setting MQTTClient to nil or freeing the binder restores it.

Thread Safety

The MQTT events can fire in a thread other than the main thread (NotifyEvents = neNoSync), and the instruments are not thread safe, so the inherited ProcessMessage runs the whole update, render and broadcast of a message inside a critical section. OnBeforeUpdate, OnAfterUpdate and OnBindingError are called inside that section, from the thread that delivered the message: do not touch VCL controls there without synchronizing, and keep the handlers short.

Requirements

The MQTT binder is available when sgcHTML, sgcHTMX and MQTT are all enabled (`SGC_HTML`, `SGC_HTMX` and `SGC_MQTT`), which is the **All-Access** edition. Unit: `sgcHTML_Instruments_MQTT`. In .NET it is not compiled in the stand-alone ASP.NET Core build of sgcHTML, which has no MQTT client; use TsgcHTMLInstrumentsBinder there.

Demo

Demos\60.HTML\01.RunTime\21.SCADA_IoT (and its form based twin in **02.DesignTime\21.SCADA_IoT**) drives a SCADA panel, gauges and a strip chart from MQTT topics through the binder.

Reference

- Properties — 3 properties (Bindings and HTMXServer inherited from TsgcHTMLInstrumentsBinder).
- Methods — SubscribeAll, plus ProcessMessage, ApplyPending and TopicMatches inherited from TsgcHTMLInstrumentsBinder.
- Events — OnBeforeUpdate, OnAfterUpdate and OnBindingError, inherited from TsgcHTMLInstrumentsBinder.

TsgcHTMLComponent_Diagram

TsgcHTMLComponent_Diagram renders a node-and-arrow flow diagram as inline SVG, where you place positioned shapes and connect them with labelled arrows.

Introduction

TsgcHTMLComponent_Diagram emits a self-contained `<svg>` drawing made of nodes and the connections between them. Each node is a positioned shape (rectangle, rounded rectangle, circle or diamond) with a text label and a theme color. Each connection draws an arrow from one node to another, optionally with its own label. You add nodes with **AddNode**, link them with **Connect**, and the component returns the complete markup through its inherited **HTML** property.

The diagram is drawn entirely with inline SVG and a small block of CSS, so it needs no JavaScript library at runtime. Connections are drawn behind the nodes and end in an arrowhead marker. Because each node is placed by its *X* and *Y* coordinates, you control the layout directly.

Quick Start

1. Drop a **TsgcHTMLComponent_Diagram** component on the form (or create it at runtime).

2. Add nodes with **AddNode** (giving each a unique id and a position) and link them with **Connect**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Diagram *oDiagram = new TsgcHTMLComponent_Diagram(NULL);
try
{
  oDiagram->AddNode("start", "Start", 340, 20, hcSuccess, nsCircle);
  oDiagram->AddNode("work", "Process", 320, 140);
  oDiagram->AddNode("done", "Done", 340, 260, hcPrimary, nsDiamond);
  oDiagram->Connect("start", "work");
  oDiagram->Connect("work", "done", "ok");
  html = oDiagram->HTML;
}
__finally
{
  delete oDiagram;
}
```

Nodes and Connections

What it does: The diagram holds two collections. *Nodes* is a list of **TsgcHTMLDiagramNode** items, each with a *NodeID*, *Text*, position (*X*, *Y*), size (*Width*, *Height*), *Color* and *Shape*. *Connections* is a list of **TsgcHTMLDiagramConnection** items, each linking a *FromNode* to a *ToNode* by id with an optional *Label_* and *Color*. Use **AddNode** and **Connect** to fill the collections at runtime.

Property	Type	Default	Description

Nodes	TsgcHTMLDiagramNodes		Collection of positioned shapes drawn in the diagram.
Connections	TsgcHTMLDiagramConnections		Collection of arrows linking nodes by id.

Canvas

What it does: *DiagramID* sets the id of the generated `<svg>`, and *DiagramWidth* and *DiagramHeight* set the SVG coordinate space the nodes are positioned within. The drawing scales to fill the width of its container while keeping that aspect ratio.

Property	Type	Default	Description
DiagramID	string	'sgcDiagram'	The id of the generated <svg> element.
DiagramWidth	Integer	800	Width of the SVG coordinate space in pixels.
DiagramHeight	Integer	400	Height of the SVG coordinate space in pixels.

Output

The inherited **HTML** property returns a `<div>` wrapping the `<svg>`, with the connections drawn first (behind the nodes) and the nodes drawn on top. The component also contributes a small block of CSS through its **GetCSS** override so the diagram follows the surrounding Bootstrap theme. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 5 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Map

TsgcHTMLComponent_Map renders an interactive Leaflet map centered on a position, with a choice of tile provider and any number of pin markers.

Introduction

TsgcHTMLComponent_Map emits a map `<div>` plus the Leaflet stylesheet, script and the initialization code needed to draw an interactive slippy map. You set the center position and zoom, pick a tile provider, then add markers (each with a latitude, longitude and optional popup and tooltip). The component builds the Leaflet JavaScript for you and returns the complete markup through its inherited **HTML** property.

The Leaflet library and its stylesheet are referenced from the unpkg CDN, and the map tiles come from the selected **TileProvider** (OpenStreetMap or a CartoDB basemap). Coordinates are always emitted with a period as the decimal separator, so the map renders correctly regardless of the locale of the server.

Quick Start

1. Drop a **TsgcHTMLComponent_Map** component on the form (or create it at runtime).

2. Set **CenterLatitude**, **CenterLongitude** and **Zoom**, then add markers with **AddMarker**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Map *oMap = new TsgcHTMLComponent_Map(NULL);
try
{
  oMap->CenterLatitude = 40.4168;
  oMap->CenterLongitude = -3.7038;
  oMap->Zoom = 13;
  oMap->AddMarker(40.4168, -3.7038, "Madrid", "Capital of Spain");
  html = oMap->HTML;
}
__finally
{
  delete oMap;
}
```

Center, Zoom and Tiles

What it does: *CenterLatitude* and *CenterLongitude* set where the map opens, *Zoom* sets the initial zoom level, and *TileProvider* selects the basemap. *Height* sets the height of the map container and *MapID* the id of its `<div>`.

Property	Type	Default	Description
CenterLatitude	Double	40.4168	Latitude the map is centered on when it opens.
CenterLongitude	Double	-3.7038	Longitude the map is centered on when it opens.

Zoom	Integer	13	Initial zoom level (higher is closer in).
TileProvider	Ts-gcHTMLMap-TileProvider	mtOpen-StreetMap	Basemap source: mtOpenStreetMap, mtCartoDB or mtCartoDBDark.
Height	string	'400px'	CSS height of the map container.
MapID	string	'sgcMap'	The id of the map element.

Markers

What it does: A map has a list of markers. Add a marker with **AddMarker**, giving its latitude and longitude and an optional popup and tooltip. Each marker is a **TsgcHTMLMapMarker** exposing *Latitude*, *Longitude*, *PopupText*, *TooltipText* and *Color*. The popup opens on click; the tooltip shows on hover.

```
oMap->AddMarker(48.8566, 2.3522, "Paris");
oMap->AddMarker(51.5074, -0.1278, "London", "Capital of the UK");
```

Output

The inherited **HTML** property returns the Leaflet stylesheet link, the map `<div>`, the Leaflet `<script>` reference and an inline initialization script that creates the map, adds the tile layer and drops the markers. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 7 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Sparkline

TsgcHTMLComponent_Sparkline renders a tiny trend chart (line, bar or area) as pure server-rendered inline SVG, with no JavaScript charting library and no CDN dependency.

Introduction

TsgcHTMLComponent_Sparkline builds the `<svg>` markup on the server. There is **no JavaScript charting library and no CDN dependency**, so the chart renders even when scripts are disabled in the browser. The component takes a list of numbers, scales them to a small drawing area, and emits a complete inline SVG through its inherited **HTML** property.

It is a tiny trend chart designed to sit inside table cells and KPI cards. The root element is a `<span>` with `display:inline-block;line-height:1;`, so a sparkline flows inline next to text without breaking the layout of a row.

Three shapes are available through **ChartType**: `slLine`, `slBar` and `slArea`. All SVG numbers are emitted locale-invariantly (always . as the decimal separator), so a machine configured with a comma decimal separator still produces valid SVG markup.

Quick Start

1. Drop a **TsgcHTMLComponent_Sparkline** component on the form (or create it at runtime).

2. Pass the series with **SetData** (or add values one by one with **AddValue**).

3. Read the **HTML** property and serve it (or drop it into a table cell, or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Sparkline *oSparkline = new TsgcHTMLComponent_Sparkline(NULL);
try
{
  oSparkline->ChartType = slLine;
  oSparkline->SetData(OPENARRAY(double, (4, 7, 5, 9, 12, 10, 15)));
  html = oSparkline->HTML;
}
__finally
{
  delete oSparkline;
}
```

When you only need the markup and no component instance, the class function **Build** does the same job in one line.

```
UnicodeString html = TsgcHTMLComponent_Sparkline::Build(
  OPENARRAY(double, (4, 7, 5, 9, 12, 10, 15)), slArea, "#198754");
```

Chart Type and Size

What it does: *ChartType* selects the shape drawn from the series, *Width* and *Height* set the size in pixels of the SVG viewport, and *LineWidth* sets the stroke thickness of the line. The `slLine` type draws a polyline, `slBar` draws one rectangle per value, and `slArea` draws the polyline plus a polygon under it filled at `fill-opacity="0.25"`.

Property	Type	Default	Description
ChartType	TsgcHTMLSparklineType	slLine	Shape drawn from the series: slLine, slBar or slArea.
Width	Integer	100	Width of the SVG in pixels.
Height	Integer	30	Height of the SVG in pixels.
LineWidth	Integer	2	Stroke width of the line, in pixels. It also drives the radius of the marker dots.

Colors and Markers

What it does: *LineColor* paints the stroke of the line, *FillColor* paints the bars and the area polygon. *ShowLastPoint* draws a dot on the final value in the line colour, and *ShowMinMax* draws two dots, the maximum in green (`#198754`) and the minimum in red (`#dc3545`).

Property	Type	Default	Description
LineColor	string	''	Colour of the line stroke and of the last-point dot. When left empty it falls back to `#0d6efd`.
FillColor	string	''	Colour of the bars and of the area polygon. When left empty it falls back to `#0d6efd`.
ShowLastPoint	Boolean	False	Draws a dot on the final value, in the line colour.
ShowMinMax	Boolean	False	Draws a green dot on the maximum and a red dot on the minimum. It only draws when there is more than one value.

Data and Scale

What it does: *Values* holds the series as a list of numbers, and the read-only *ValueCount* and *ValueItem* let you inspect it. *Min* and *Max* pin the vertical scale, and a value of zero means auto-scale from the series itself. Fill the series with **AddValue**, **SetData** or **LoadFromDataSet**, and empty it with **ClearData**. The class function **Build** returns the markup in one line without an instance.

When both *Min* and *Max* are zero the component reads the smallest and the largest value in the series and scales to that range. Set either one to a non-zero value to pin that end of the scale, which is what you want when several sparklines in a table must be comparable. If the series is flat or holds a single value, the component builds a synthetic range around the value, so nothing divides by zero and the line renders centred.

A bound is honoured when **either its Auto flag is False or the value is non-zero**. So a Min of zero means auto-scale only while *AutoMin* is left at its True default. Set *AutoMin* to False and Min is taken verbatim, which is the only way to pin a floor of exactly 0, the natural baseline of a bar sparkline. *AutoMax* does the same for the top of the scale. Code written before the flags existed is unaffected, because a non-zero bound is still honoured with the flag left at True.

Property	Type	Default	Description
Values	TStringList	empty list	The series, one number per line. Parsing accepts both `.` and `,` as decimal separator.
Min	Double	0	Lower bound of the vertical scale. Zero means auto-scale while AutoMin is True.
Max	Double	0	Upper bound of the vertical scale. Zero means auto-scale while AutoMax is True.
AutoMin	Boolean	True	False pins the bottom of the scale to Min, a floor of exactly 0 included.
AutoMax	Boolean	True	False pins the top of the scale to Max, a ceiling of exactly 0 included.
ValueCount	Integer	read-only	Number of values currently in the series.
ValueItem	Double	read-only	Indexed read-only access to a single value of the series, parsed as a number.

Output

The inherited **HTML** property returns a `<span>` wrapping a complete inline `<svg>`. There is no script tag and no library reference. Serve it directly, drop it into a table cell, or assign it to a page template's BodyContent.

The `<span>` carries the inherited **ElementID**, **CSSClass**, **Style** and **Attributes**, so you can target it from your own stylesheet. Because the SVG is built server-side and the colours are attribute-encoded, the markup is safe to embed as-is.

Reference

- Properties — 15 properties available on the component.
- Methods — 5 public methods.
- Events — this component has no published events of its own.

TsgcHTMLComponent_Heatmap

TsgcHTMLComponent_Heatmap renders a matrix heatmap as pure server-rendered inline SVG, colouring every cell by where its value falls in the value range.

Introduction

TsgcHTMLComponent_Heatmap draws a grid of rows and columns where the colour of each cell encodes a numeric value. You declare the row and column labels, then set one value per grid position. The component works out the value range, interpolates a colour for every cell, and returns the finished markup through its inherited **HTML** property.

The heatmap is **pure server-rendered inline SVG**. There is no JavaScript charting library and no Content Delivery Network dependency. The component builds the complete `<svg>` markup on the server, so the heatmap renders even when scripts are disabled in the browser. Nothing is downloaded, nothing is executed, the picture is already in the HTML you send.

Quick Start

1. Drop a **TsgcHTMLComponent_Heatmap** component on the form (or create it at runtime).

2. Fill **RowLabels** and **ColumnLabels**, then set the values with **SetCell**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Heatmap *oHeatmap = new TsgcHTMLComponent_Heatmap(NULL);
try
{
  oHeatmap->RowLabels->Add("Monday");
  oHeatmap->RowLabels->Add("Tuesday");
  oHeatmap->ColumnLabels->Add("09h");
  oHeatmap->ColumnLabels->Add("10h");
  oHeatmap->ColumnLabels->Add("11h");
  oHeatmap->SetCell(0, 0, 12);
  oHeatmap->SetCell(0, 1, 48);
  oHeatmap->SetCell(0, 2, 30);
  oHeatmap->SetCell(1, 0, 5);
  oHeatmap->SetCell(1, 2, 64);
  oHeatmap->ShowValues = true;
  oHeatmap->ShowLegend = true;
  html = oHeatmap->HTML;
}
__finally
{
  delete oHeatmap;
}
```

Grid and Labels

What it does: *RowLabels* and *ColumnLabels* define the shape of the matrix, one entry per row and one per column. *ShowRowLabels* and *ShowColumnLabels* turn each label strip on or off. *CellSize* and *CellGap* control the size of the squares and the space between them, *Rounded* gives them rounded corners, and *FontSize* and *LabelColor* style the label text.

Cells are addressed by zero-based row and column index. A cell may reference a row or a column beyond the declared labels, and the grid grows to fit it, so the labels are a naming convenience rather than a hard bound. The width reserved for the row label column is estimated from the longest label, because there are no font metrics available on the server, and it is capped at 200 pixels.

Property	Type	Default	Description
RowLabels	TStringList	empty	One label per row, in row order. Row index 0 is the first entry.
ColumnLabels	TStringList	empty	One label per column, in column order. Column index 0 is the first entry.
ShowRowLabels	Boolean	True	Draws the row labels to the left of the grid.
ShowColumnLabels	Boolean	True	Draws the column labels above the grid.
CellSize	Integer	40	Width and height of every cell, in pixels.
CellGap	Integer	2	Gap between adjacent cells, in pixels.
Rounded	Boolean	False	Draws the cells with rounded corners.
FontSize	Integer	11	Font size in pixels for labels, cell values and legend text.
LabelColor	string	”	Colour of the row, column and legend labels. Empty falls back to `#6c757d` .

Colour Scale

What it does: The colour of a cell is a **linear RGB interpolation between *MinColor* and *MaxColor***, based on where the cell value falls in the value range. A value equal to the minimum gets MinColor, a value equal to the maximum gets MaxColor, everything in between gets a blend of the two channels by channel.

Both colours must be plain hex (`#rrggbb` or `#rgb`) for the interpolation to work, because the component has to read the red, green and blue channels out of them. A colour that is not plain hex, a CSS colour name such as `red` or a custom property such as `var(--brand)`, cannot be interpolated. In that case the component falls back to whichever endpoint colour is nearer, so the grid degrades to two colours instead of a smooth ramp.

MinValue and *MaxValue* pin the ends of the scale. Left at 0 they mean auto, and the range is taken from the data. Set them when several heatmaps must share one scale so that the same colour means the same number in every picture. If the data is flat, every value is the same, there is no range to scale against, so the component uses the middle of the colour ramp instead of dividing by zero. A grid position with no cell is painted *EmptyColor*, which is how a hole in the matrix is told apart from a low value.

A bound is honoured when **either its Auto flag is False or the value is non-zero**. So a MinValue of zero means auto-scale only while *AutoMin* is left at its True default. Set *AutoMin* to False and MinValue is taken verbatim, which is the only way to pin the bottom of the ramp to exactly 0, so that a zero cell always gets MinColor whatever the data reaches down to today. *AutoMax* does the same for the top of the ramp. Code written before the flags existed is unaffected, because a non-zero bound is still honoured with the flag left at True.

ShowLegend draws a MinColor to MaxColor gradient bar under the grid, with the minimum value as the left end label and the maximum value as the right end label. Each component gets its own gradient id, taken from its *Name*,

or from a process-wide counter when it has none, so two heatmaps on one page never share a gradient by accident.

Property	Type	Default	Description
MinColor	string	''	Colour of the lowest value. Empty falls back to `#e3f2fd`. Must be plain hex to interpolate.
MaxColor	string	''	Colour of the highest value. Empty falls back to `#0d47a1`. Must be plain hex to interpolate.
EmptyColor	string	''	Colour of a grid position that has no cell. Empty falls back to `#f1f3f5`.
MinValue	Double	0	Value mapped to MinColor. 0 means auto while AutoMin is True, the lowest value in the data.
MaxValue	Double	0	Value mapped to MaxColor. 0 means auto while AutoMax is True, the highest value in the data.
AutoMin	Boolean	True	False pins the bottom of the colour scale to MinValue, a floor of exactly 0 included.
AutoMax	Boolean	True	False pins the top of the colour scale to MaxValue, a ceiling of exactly 0 included.
ShowLegend	Boolean	False	Draws a gradient bar under the grid with the min and max values as end labels.

Values and Data

What it does: *Cells* is the collection of values, each one a **TsgcHTMLHeatmapCell** with a *Row*, a *Col* and a *Value*. Set a value with **SetCell**, look one up with **FindCell**, and empty the collection with **ClearData**. **LoadFromDataSet** fills the whole component from a dataset in one call, building the row and column labels from the distinct field values as it goes.

ShowValues prints the number inside each cell and *Decimals* says how many decimal places to use. The **text auto-contrasts by cell luminance**: the component computes the perceived luminance of the interpolated background and prints the number dark (`#212529`) on light cells and light (`#ffffff`) on dark cells. You never have to pick a text colour, and the numbers stay readable at both ends of the ramp. Values are only drawn when the cell is large enough to hold them, that is when CellSize is at least FontSize plus 4.

Every cell also carries an SVG `<title>` with its raw value, so the browser shows the number as a native tooltip when the pointer rests on the cell, without any script.

Property	Type	Default	Description
Cells	TsgcHTMLHeatmapCells	empty	Collection of cells. Public, not published. Each item exposes Row, Col and Value.

ShowVal-ues	Boolean	False	Prints the value inside each cell, with an auto-contrasting text colour.
Decimals	Integer	0	Decimal places used for the printed values and the leg-end end labels.

```
// Row field, column field, value field. Labels are built automatically.
oHeatmap->LoadFromDataSet(qrySales, "Region", "Month", "Amount");
```

Output

The inherited **HTML** property returns a `<div>`, carrying the ElementID, CSSClass, Style and Attributes of the component, wrapping one complete inline `<svg>` element. The SVG holds the label texts, one `<rect>` per grid position, the optional value texts and the optional legend. There is no script tag and no external reference, so the string is self-contained. Serve it directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 20 properties available on the component.
- Methods — 4 public methods.
- Events — this component has no published events of its own.

TsgcHTMLComponent_TreeMap

Renders a treemap as a single inline SVG image, where every rectangle's area is proportional to its value.

Introduction

A treemap shows a set of values as nested rectangles that fill a fixed canvas. The bigger the value, the bigger the rectangle. It is the fastest way to see, in one picture, which few items dominate a total and which are noise, for example sales by product, disk usage by folder, or budget by department.

This component is a pure server-rendered inline SVG treemap. There is no JavaScript charting library and no CDN dependency. The `<svg>` markup is built on the server by the component and sent inside the page. Nothing is downloaded at runtime, nothing is drawn by a script, and the treemap still renders when the browser has scripts disabled. It also works in an email client, in a PDF pipeline, or anywhere else that accepts plain HTML.

The layout uses the **squarified** algorithm of Bruls, Huizing and van Wijk. Rectangles come out close to square, so their sizes are easy to compare by eye. A naive treemap slices the canvas alternately in one direction and produces long thin stripes that are almost impossible to read. This one does not. On top of that, **areas are exactly proportional to values**: a rectangle's area equals that item's share of the total, so the picture never lies about the data.

Quick Start

1. Create the component and set the canvas size with *Width* and *Height*.

2. Add the data with *AddItem*, once per rectangle. The order does not matter, the component sorts the items internally.

3. Read the inherited *HTML* property and write it into your page.

```
TsgcHTMLComponent_TreeMap *oTreeMap = new TsgcHTMLComponent_TreeMap(NULL);
try
{
  oTreeMap->Width = 800;
  oTreeMap->Height = 500;
  oTreeMap->ColorScheme = tmBlue;
  oTreeMap->ShowValues = true;

  oTreeMap->AddItem("Servers", 4200);
  oTreeMap->AddItem("Storage", 2600);
  oTreeMap->AddItem("Network", 1800);
  oTreeMap->AddItem("Licenses", 950);
  oTreeMap->AddItem("Support", 400);

  UnicodeString vHTML = oTreeMap->HTML;
}
__finally
{
  delete oTreeMap;
}
```

Layout and Size

What it does: the treemap is drawn inside a canvas of *Width* by *Height* pixels, and that whole canvas is always filled. The rectangles are placed by the squarified algorithm of Bruls, Huizing and van Wijk: the items are sorted descending by value, then grouped into rows, and each row is laid out along the shortest side of the space that is still free. A row keeps growing while the aspect ratio of its worst rectangle improves, and is closed as soon as adding one more item would make it worse. The result is a set of rectangles that are as close to square as the data allows, instead of the long thin stripes a plain slice-and-dice treemap produces.

Whatever the shape, **the area of each rectangle is exactly its share of the total**. The values are scaled once so that their sum equals *Width* multiplied by *Height*, and the layout preserves that area for every item. An item worth 25 percent of the total occupies 25 percent of the canvas, always.

Padding is emitted as the stroke width of every rectangle, drawn in *BackgroundColor*, so it reads as a clean gap between the tiles. *MinLabelSize* is the size a rectangle must reach before it is allowed to carry a label.

Property	Type	Default	Description
Width	Integer	600	Width of the SVG canvas in pixels.
Height	Integer	400	Height of the SVG canvas in pixels.
Padding	Integer	2	Stroke width drawn around every rectangle in the background colour, which reads as the gap between tiles.
MinLabelSize	Integer	40	Minimum width and height, in pixels, a rectangle must have before its label is drawn.

Colors

What it does: *ColorScheme* selects one of six built-in palettes. Each palette holds 10 colours which are applied to the rectangles in order, and the palette repeats from the start once there are more than 10 items. An item can override this: if you set the *Color* of an item in *Items*, that colour always wins over the scheme. *BackgroundColor* paints the gap between the tiles, because it is the colour used for the rectangle strokes.

The available schemes are **tmDefault** (a mixed, high-contrast palette), **tmBlue**, **tmGreen**, **tmWarm** (oranges and reds), **tmCool** (blues and purples) and **tmGray**.

Property	Type	Default	Description
ColorScheme	TsgcHTMLTreeMapScheme	tmDefault	Built-in 10-colour palette applied to the rectangles in order, repeating past 10 items.
BackgroundColor	string	''	Colour of the gaps between the tiles. When left empty it falls back to `#ffffff`.

Labels and Values

What it does: *ShowLabels* draws the item label inside its rectangle, and *ShowValues* adds the value on a second line under the label. A label is only drawn when the rectangle is at least *MinLabelSize* wide **and** at least *MinLabelSize* tall, so the small tiles stay clean instead of overflowing with clipped text. The value line needs a little more height still, the rectangle must also be at least *MinLabelSize* plus *FontSize* tall.

The text colour is picked automatically for each tile from the perceived luminance of its fill, dark `#212529` on light tiles and white on dark tiles, so the label stays readable on every palette. *FontSize* sets the label size in pixels, the value line is drawn one pixel smaller. *Decimals* controls how many decimal places the value is printed with.

Even when a tile is too small for a label, it is not silently anonymous. Every rectangle carries an SVG `<title>` element with the text "label: value", so hovering it shows a native browser tooltip with no script involved.

Property	Type	Default	Description
ShowLabels	Boolean	True	Draws the item label inside rectangles that are big enough.
ShowValues	Boolean	False	Draws the value under the label, on rectangles tall enough to hold both lines.
FontSize	Integer	12	Label font size in pixels. The value line uses one pixel less.
Decimals	Integer	2	Decimal places used when the value is printed, in the label and in the tooltip.

Data

What it does: the data lives in the *Items* collection. Each item has a *Label_* (the trailing underscore is there because `Label` is a reserved word), a *Value*, an optional *Color*, and an optional *ID* and *ParentID* pair used for grouping. Fill the collection at design time in the Object Inspector, or at run time with *AddItem*. *ClearData* empties it, *LoadFromDataSet* fills it from any `TDataSet`, and the class function *Build* renders a treemap from two plain arrays without creating an instance at all.

Two rules apply to every item before layout. Items are **sorted descending by value** internally, because the squarified algorithm needs them in that order, so the order you add them in does not change the picture. And items whose value is **zero or less are dropped**, because an area cannot be given to them.

Two levels are supported. Set *ID* on an item to make it a parent, and set *ParentID* on other items to attach them to it. A parent is drawn faded, with a fill opacity of 0.35, and its children are squarified inside it, under a strip left free at the top for the parent label. A parent whose own *Value* is 0 takes the sum of its children as its value, so you can declare groups without having to add up their totals yourself.

Member	Type	Default	Description
Items	TsgcHTMLTreeMapItems	empty	Collection of the rectangles to draw.
AddItem	method		Adds one item with a label, a value and an optional colour, and returns it.
ClearData	method		Removes every item from the collection.

LoadFrom-DataSet	method		Replaces the items with one item per record of a dataset.
Build	class method		Renders a treemap straight from an array of labels and an array of values.

```
// a two-level treemap: 2 groups, 5 leaves
oTreeMap->Items->Add()->ID = "eu";
oTreeMap->Items->Items[0]->Label_ = "Europe";

oTreeMap->AddItem("Spain", 1200)->ParentID = "eu";
oTreeMap->AddItem("Germany", 2100)->ParentID = "eu";
oTreeMap->AddItem("France", 1500)->ParentID = "eu";

oTreeMap->Items->Add()->ID = "us";
oTreeMap->Items->Items[4]->Label_ = "America";

oTreeMap->AddItem("USA", 3400)->ParentID = "us";
oTreeMap->AddItem("Canada", 900)->ParentID = "us";
```

Output

The inherited **HTML** property returns a `<div>` that wraps one complete inline `<svg>` element. The `<div>` carries the inherited **ElementID**, **CSSClass**, **Style** and **Attributes**, so you can position and style it like any other block in your page. The `<svg>` holds one `<rect>` per item, each with its own `<title>` tooltip, plus the `<text>` label and value nodes.

There is no `<script>` tag, no external stylesheet and no image request. Every label and every colour is HTML-encoded on the way out, and all the SVG numbers are emitted locale-invariantly, always with . as the decimal separator, so the markup stays valid on a machine whose locale uses a comma.

When the collection is empty, or when every value is zero or less, the component still returns a valid but empty `<svg>` of the requested size rather than failing.

Reference

- Properties — 11 properties available on the component.
- Methods — 4 public methods.
- Events — this component has no published events of its own.

TsgcHTMLComponent_CandlestickChart

TsgcHTMLComponent_CandlestickChart renders an OHLC candlestick stock chart as pure server-rendered inline SVG, with an optional volume strip.

Introduction

TsgcHTMLComponent_CandlestickChart builds a candlestick (OHLC) price chart entirely on the server. There is **no JavaScript charting library and no CDN dependency**. The component composes the `<svg>` markup itself and returns it through its inherited **HTML** property, so the chart renders even when scripts are disabled in the browser.

You feed the chart a list of points, where every point carries a label plus its Open, High, Low and Close prices and an optional volume. Each point is drawn as a *body* (the rectangle spanning open to close) and a *wick* (the vertical line spanning low to high). A candle is drawn in **UpColor** when Close is greater than or equal to Open, otherwise in **DownColor**.

Every candle also carries an SVG `<title>` of the form `label O:.. H:.. L:.. C:..`, so the browser shows the OHLC values as a native tooltip when the user hovers a candle. No script is involved in that either.

Quick Start

1. Drop a **TsgcHTMLComponent_CandlestickChart** component on the form (or create it at runtime).

2. Add the OHLC points with **AddPoint**, one call per bar.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_CandlestickChart *oChart = new TsgcHTMLComponent_CandlestickChart(NULL);
try
{
  oChart->Width = 800;
  oChart->Height = 400;
  oChart->ShowVolume = true;
  oChart->AddPoint("Mon", 101.20, 104.80, 100.50, 104.10, 1250000);
  oChart->AddPoint("Tue", 104.10, 105.60, 102.90, 103.05, 980000);
  oChart->AddPoint("Wed", 103.05, 107.40, 102.80, 107.00, 1640000);
  oChart->AddPoint("Thu", 107.00, 107.90, 105.10, 105.40, 1120000);
  oChart->AddPoint("Fri", 105.40, 109.30, 105.20, 109.00, 1875000);
  html = oChart->HTML;
}
__finally
{
  delete oChart;
}
```

Candles and Colours

What it does: Every point becomes one candle: the body is the open-to-close rectangle and the wick is the vertical high-to-low line. *UpColor* paints the candles where Close is greater than or equal to Open, *DownColor* paints the rest. *WickColor* left empty means each wick takes its own candle's body colour, so up wicks are green and down wicks are red. Set it to draw every wick in one fixed colour. *CandleWidth* controls how wide the bodies are.

A body that would be thinner than 1 pixel is clamped to 1 pixel, so a flat candle where open equals close is still visible as a thin line.

Property	Type	Default	Description
UpColor	string	''	Colour of the candles that closed at or above their open. Empty falls back to #198754 (green).
DownColor	string	''	Colour of the candles that closed below their open. Empty falls back to #dc3545 (red).
WickColor	string	''	Colour of the high-to-low wick. Empty means each wick uses its own candle's body colour.
CandleStyle	TsgcHTMLCandlestickStyle	cssCandle	cssCandle draws candlesticks, cssOHLC draws OHLC bars (high-low line, left tick at the open, right tick at the close).
CandleWidth	Integer	0	Body width in pixels. 0 means auto, where each candle takes 60% of its slot. Any value is capped so candles cannot overlap.

Axis, Grid and Scale

What it does: *ShowGrid* draws the horizontal gridlines, *ShowAxis* draws the axis lines and all the labels. *GridLines* sets how many horizontal lines are spread across the price area, each one labelled with its price on the left axis. GridLines must be greater than 1 for anything to be drawn. *GridColor* and *AxisColor* tint them, *FontSize* sizes the label text and *Decimals* sets how many decimal places the price labels and the candle tooltips show.

X labels are thinned automatically so they never overlap. The component works out how many labels fit across the plot area (about one label per 70 pixels) and draws every Nth label, always including the last point. A long series therefore never produces a smeared axis.

MinValue and *MaxValue* left at 0 mean auto-scale from the data, and the scan covers the Low, High, Open and Close of every point. Set them to pin the price axis to a fixed range. A flat series or a single point gets a synthetic range, so the chart never divides by zero.

A bound is honoured when **either its Auto flag is False or the value is non-zero**. So a MinValue of zero means auto-scale only while *AutoMin* is left at its True default. Set *AutoMin* to False and MinValue is taken verbatim, which is the only way to anchor the price axis at exactly 0. *AutoMax* does the same for the top of the axis. Code written before the flags existed is unaffected, because a non-zero bound is still honoured with the flag left at True.

All the numbers written into the SVG are emitted locale-invariantly, always with . as the decimal separator, so the markup stays valid on a machine whose locale uses a comma.

Property	Type	Default	Description
ShowAxis	Boolean	True	Draws the X and Y axis lines plus the price and label text.
ShowGrid	Boolean	True	Draws the horizontal gridlines behind the candles.
GridColor	string	''	Colour of the gridlines. Empty falls back to #e9ecef.

AxisColor	string	''	Colour of the axis lines and the label text. Empty falls back to #6c757d.
GridLines	Integer	5	Number of horizontal gridlines and price labels. Must be greater than 1.
MinValue	Double	0	Lowest price on the Y axis. 0 means auto-scale while AutoMin is True.
MaxVal-ue	Double	0	Highest price on the Y axis. 0 means auto-scale while AutoMax is True.
AutoMin	Boolean	True	False pins the bottom of the price axis to MinValue, a floor of exact-ly 0 included.
AutoMax	Boolean	True	False pins the top of the price axis to MaxValue, a ceiling of exactly 0 included.
FontSize	Integer	10	Font size in pixels of the axis labels.
Decimals	Integer	2	Decimal places used for the price labels and the candle tooltips.

Volume

What it does: When *ShowVolume* is True the price area shrinks to 75% of the plot height and a volume bar strip is drawn in the bottom 20%. Each bar is scaled against the largest volume in the series and is tinted with its own candle's colour at 45% opacity, so the volume strip reads as up days and down days at a glance. Points whose vol-ume is zero simply produce no bar.

Property	Type	Default	Description
ShowVolume	Boolean	False	Draws a volume bar strip below the price area.

Data

What it does: The chart holds a *Points* collection of **TsgcHTMLCandlestickPoint** items. Each item exposes *La-bel_* (the name has a trailing underscore because `Label` is a reserved word), plus *Open*, *High*, *Low*, *Close* and *Vol-ume*. Add points one by one with **AddPoint**, empty the collection with **ClearData**, or fill it from a dataset with **Load-FromDataSet**. *Width* and *Height* size the generated SVG.

With an empty Points collection the component still emits a valid SVG that contains the axis and the grid, so a chart with no data yet never breaks the page.

Property	Type	Default	Description

Points	TsgcHTMLCandlestick-Points	empty	The OHLC points drawn by the chart, one candle per item.
Width	Integer	800	Width of the generated SVG in pixels.
Height	Integer	400	Height of the generated SVG in pixels.

```
oChart->LoadFromDataSet(qryPrices, "TradeDate", "Open", "High", "Low", "Close", "Volume");
```

What's New

The candlestick chart gained an OHLC bar style, technical indicators, an interactive crosshair and live ticks pushed from the server. The new properties default to the previous behaviour, so an existing chart renders exactly the same HTML. The runtime demo **60.HTML\01.RunTime\15.Reports** (sgcReportsServer, *Analytics* page) shows a candlestick chart with SMA, EMA and Bollinger bands, the crosshair and live ticks.

Indicators

The Indicators collection adds lines computed on the server from the Close prices: `cikSMA` (simple moving average), `cikEMA` (exponential moving average, seeded with the SMA of the first period) and `cikBollinger` (middle SMA plus upper and lower bands at StdDevs population standard deviations, with a translucent fill). Each item has *Kind*, *Period* (20), *StdDevs* (2), *Color*, *LineWidth* (1) and *Caption*. A legend is drawn at the top left, and the automatic price scale includes the indicator values.

OHLC Bars and Crosshair

CandleStyle switches between candlesticks (`cssCandle`) and OHLC bars (`cssOHLC`). ShowCrosshair adds a small script that draws guide lines snapped to the nearest candle and a readout with the label, Open, High, Low, Close and Volume while the mouse or a finger moves over the chart. ChartID is the id of the chart; when empty it is ElementID, or an id generated once per instance.

Live Ticks (Server Push)

GetLiveTickFragment applies a tick to the chart: it appends a new candle, or updates the last one when `aReplaceLast` is True (the current candle is still open), trims *Points* to LiveMaxPoints and returns the whole chart with `hx-swap-oob="outerHTML"`, ready to replace the chart in the browser. Keep the same instance alive between ticks and give it a stable ChartID before the first render.

Output

The inherited **HTML** property returns a `<div>` wrapping a complete inline `<svg>`. The div carries the ElementID, CSS-Class, Style and Attributes you set on the component. Unless **ShowCrosshair** is True there is no script tag, and there is never an external reference of any kind, so you can serve the string directly, embed it in an email or assign it to the BodyContent of a page template.

Reference

- Properties — 24 properties available on the component.
- Methods — 4 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_QRCode

TsgcHTMLComponent_QRCode encodes text into a QR Code and renders it as a dependency-free, inline SVG image, computed entirely on the server in pure Pascal.

Introduction

TsgcHTMLComponent_QRCode wraps a complete ISO/IEC 18004 QR Code encoder (`TsgcHTMLQRCodeEncoder`, used internally). The encoder chooses the cheapest of the numeric, alphanumeric and byte modes for the **Data** you assign, selects the smallest symbol version from 1 to 40 that satisfies the requested **ECCLevel**, and applies Reed–Solomon error correction over GF(256) with per-version block splitting and codeword interleaving. It then evaluates all 8 standard mask patterns against the 4 penalty rules from the specification and picks the best one, and writes the BCH format (15,5) and version (18,6) information into the symbol. The resulting grid is emitted as compact SVG markup, one `<rect>` per horizontal run of dark modules rather than one per module.

Dependency-free by design. The QR symbol is computed on the server in pure Pascal and emitted as inline SVG. There is no JavaScript, no canvas, no external library, no Content Delivery Network and no image file involved. The markup therefore prints correctly, works with JavaScript disabled in the browser, can be embedded directly in an email or a PDF, and costs one extra HTTP request, exactly zero, because the SVG is already part of the page.

Quick Start

1. Drop a **TsgcHTMLComponent_QRCode** component on the form (or create it at runtime).

2. Set **Data** to the text to encode and, optionally, **ECCLevel**, **ModuleSize** and **Caption**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_QRCode *oQRCode = new TsgcHTMLComponent_QRCode(NULL);
try
{
  oQRCode->Data = "https://www.esegece.com";
  oQRCode->ECCLevel = qrHigh;
  oQRCode->ModuleSize = 5;
  oQRCode->Caption = "Scan to visit our website";
  html = oQRCode->HTML;
}
__finally
{
  delete oQRCode;
}
```

Data and Error Correction

What it does: *Data* is the text to encode (a URL, a vCard, a Wi–Fi credential string, or plain text). *ECCLevel* selects how much of the symbol can be lost, to dirt, a print defect or a logo overlay, and still be recovered. The encoder always picks the smallest QR version, from 1 to 40, that fits the data at the requested level, so a higher *ECCLevel* produces a bigger symbol for the same *Data*.

Property	Type	Default	Description
Data	string	”	The text to encode.
ECCLevel	TsgcHTM-LQRECCLevel	qrMedi-um	Error correction level: qrLow, qrMedium, qrQuartile or qrHigh.

Level	Recoverable	Typical use
qrLow	≈ 7%	Clean digital display, maximum data capacity, smallest symbol.
qrMedi-um	≈ 15%	General purpose, the default.
qrQuar-tile	≈ 25%	Printed material handled by hand, moderate wear.
qrHigh	≈ 30%	Symbol carries a logo overlay, or is printed small, outdoors, or on a poor surface.

Size and Quiet Zone

What it does: *ModuleSize* is the pixel size of one module (one square “dot” of the symbol). *QuietZone* is the blank margin around the symbol, expressed in modules; the ISO/IEC 18004 specification requires at least 4. The final SVG is square, and its side in pixels is `(Size + 2 * QuietZone) * ModuleSize`, where *Size* is the module count of the version the encoder selected.

Property	Type	Default	Description
ModuleSize	Inte-ger	4	Pixel size of one module. Values below 1 are clamped to 1.
QuietZone	Inte-ger	4	Margin in modules around the symbol. Negative values are clamped to 0.

Colors and Caption

What it does: *DarkColor* and *LightColor* set the module and background colors of the symbol. *Caption* is an optional line of text placed under the symbol, shown when *CaptionVisible* is True and *Caption* is not empty; it is HTML-encoded automatically.

Property	Type	Default	Description

DarkColor	string	#000000	Color of the dark modules. An empty value falls back to #000000.
LightColor	string	#ffffff	Background color of the symbol. An empty value falls back to #ffffff.
Caption	string	”	Optional text shown under the symbol, HTML-encoded.
CaptionVis-ible	Boolean	True	Shows the caption when True and Caption is not empty.

Server-Side SVG, No JavaScript

Call **GetSVG** when you only want the bare `<svg></svg>` element, for example to write a standalone .svg file, to embed the symbol in a PDF, or to place it inside markup you build yourself. Use the class function **Build** for a one-line helper that creates a temporary component, renders it and frees it, useful for a single quick call without managing a component instance.

```
// one-line helper, no component instance to manage
html = TsgcHTMLComponent_QRCode::Build("https://www.esegece.com", qrHigh, 5);
```

Output

The inherited **HTML** property returns a `<div class="d-inline-block text-center">`, carrying the component's **ElementID**, that wraps the inline `<svg>` and, when applicable, the caption `<div class="small text-muted">`. The `<svg>` itself carries `shape-rendering="crispEdges"`, so modules stay sharp at any zoom level, and `role="img"` for accessibility.

Reference

- Properties — 8 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Barcode

TsgcHTMLComponent_Barcode encodes text into a Code 128 barcode and renders it as a dependency-free, inline SVG image, computed entirely on the server in pure Pascal.

Introduction

TsgcHTMLComponent_Barcode implements a pure Pascal Code 128 encoder. It picks the START subset automatically and switches between subsets A, B and C on the fly to keep the symbol as short as possible: runs of digits are packed two per symbol in subset C, control characters use subset A, and everything else uses subset B. The encoder appends the mandatory modulo-103 checksum symbol and the stop pattern, and lays out a 10-module quiet zone on each side of the bars.

Dependency-free by design. The barcode is computed on the server in pure Pascal and emitted as inline SVG. There is no JavaScript, no canvas, no external library, no Content Delivery Network and no image file involved. The markup therefore prints correctly, works with JavaScript disabled in the browser, can be embedded directly in an email or a PDF, and costs one extra HTTP request, exactly zero, because the SVG is already part of the page.

Quick Start

1. Drop a **TsgcHTMLComponent_Barcode** component on the form (or create it at runtime).

2. Set **Data** to the ASCII text to encode and, optionally, **ModuleWidth**, **Height** and **ShowText**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Barcode *oBarcode = new TsgcHTMLComponent_Barcode(NULL);
try
{
  oBarcode->Data = "SGC-4821057";
  oBarcode->ModuleWidth = 2;
  oBarcode->Height = 60;
  oBarcode->ShowText = true;
  html = oBarcode->HTML;
}
__finally
{
  delete oBarcode;
}
```

Data and Code 128 Subsets

What it does: *Data* is the text to encode; *Symbology* identifies the barcode standard, currently always Code 128. Data must be ASCII, character codes 0 to 127. The encoder switches between Code 128 subsets A, B and C as it scans the text: digit pairs are packed into subset C for density, control characters use subset A, and printable text uses subset B, so the same component produces a short symbol whether Data is numeric, alphanumeric or mixed. A modulo-103 checksum symbol and the stop pattern are appended automatically; you never add them yourself.

A character above code point 127 raises an exception at render time naming the offending character code and its position in the string, so validate or transliterate the text before assigning it to Data. An empty Data renders an empty string, no markup at all.

Property	Type	Default	Description
Data	string	''	The ASCII text to encode.
Symbology	TsgcHTMLBar-codeSymbology	bsCode128	Barcode standard. Only bsCode128 is implemented to-day.

Size and Text

What it does: *ModuleWidth* is the width, in pixels, of the narrowest bar; every other bar and space is a whole multiple of it. *Height* is the height of the bars in pixels. *ShowText* prints the human-readable Data string under the bars.

Property	Type	Default	Description
Mod-uleWidth	Integer	2	Width in pixels of the narrowest bar. Values below 1 are clamped to 1.
Height	Integer	60	Bar height in pixels. Values below 1 are clamped to 1.
ShowText	Boolean	True	Prints the human-readable text under the bars.

Widen **ModuleWidth** for low-resolution printers or for symbols meant to be scanned from a long distance. Keep enough contrast between **Color** and **BackgroundColor**, and do not scale the rendered SVG non-uniformly, or scanners will fail to read it.

Colors

What it does: *Color* sets the bar (and text) color; *BackgroundColor* sets the background behind the bars.

Property	Type	Default	Description
Color	string	#000000	Bar and text color.
BackgroundColor	string	#ffffff	Background color behind the bars.

Server-Side SVG, No JavaScript

Call **GetSVG** when you only want the bare `<svg></svg>` element, for example to write a standalone .svg file, to embed the symbol in a PDF, or to place it inside markup you build yourself. GetSVG validates Data first and raises an exception if it contains a non-ASCII character, so you can call it inside a try block to surface bad input early.

```
UnicodeString svg = oBarcode->GetSVG();
```

Output

The inherited **HTML** property returns a `<div class="sgc-barcode d-inline-block">` wrapping the inline `<svg>`. The `<svg>` carries `role="img"` and an `aria-label` equal to Data for accessibility.

Reference

- Properties — 7 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_NavBar

TsgcHTMLComponent_NavBar renders a responsive Bootstrap navigation bar with an optional brand, a collapsible toggler and a collection of navigation links.

Introduction

TsgcHTMLComponent_NavBar emits a Bootstrap `<nav class="navbar">` block: a brand on the left, a hamburger toggler that collapses the links on small screens, and an unordered list of `nav-item` links built from the **Items** collection. You pick a light or dark **Theme**, choose the breakpoint at which the bar stops collapsing with **Expand**, and read the finished markup through the inherited **HTML** property.

Each link is a **TsgcHTMLNavItem** with its own *Text*, *Href*, *Active* and *Disabled* flags, so the bar is described entirely as data and no hand-written HTML is needed.

Quick Start

1. Drop a **TsgcHTMLComponent_NavBar** component on the form (or create it at runtime).

2. Set the **Brand**, pick a **Theme**, then add one item per link to the **Items** collection.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_NavBar *oNavBar = new TsgcHTMLComponent_NavBar(NULL);
try
{
  oNavBar->Brand = "Acme";
  oNavBar->Theme = ntDark;
  oNavBar->Expand = neLarge;
  TsgcHTMLNavItem *item;
  item = oNavBar->Items->Add(); item->Text = "Home"; item->Href = "/"; item->Active = true;
  item = oNavBar->Items->Add(); item->Text = "Products"; item->Href = "/products";
  item = oNavBar->Items->Add(); item->Text = "Contact"; item->Href = "/contact";
  html = oNavBar->HTML;
}
__finally
{
  delete oNavBar;
}
```

Appearance

What it does: *Brand* and *BrandHref* set the masthead text and its link. *Theme* chooses a light or dark bar, and *Expand* selects the breakpoint at which the links stop collapsing into the toggler. *Fluid* switches between a full-width and a centred container, while *CSSClass* appends extra Bootstrap utility classes.

Property	Type	Default	Description
Brand	string	''	Brand text shown at the start of the bar.

BrandHref	string	'#'	Link target for the brand.
Theme	TsgcHTML-NavBarTheme	ntDark	Light (ntLight) or dark (ntDark) color scheme.
Expand	TsgcHTML-NavBarExpand	neLarge	Breakpoint where links stop collapsing: neSmall, neMedium, neLarge, neXLarge or neAlways.
Fluid	Boolean	True	True uses a full-width container-fluid, False a centred container.
NavBarID	string	''	Id of the generated nav element; defaults to sgcNavBar when empty.
CSSClass	string	''	Extra CSS classes appended to the nav element.

Items

What it does: The **Items** collection holds the navigation links. Call **Items.Add** to append a link, then set its sub-properties. Each item is a **TsgcHTMLNavItem** exposing the following published properties.

Item Property	Type	Default	Description
Text	string	''	Visible caption of the link (HTML-encoded on output).
Href	string	''	Link target written to the anchor's href attribute.
Active	Boolean	False	Marks the link as the current page (adds the active class).
Disabled	Boolean	False	Renders the link as disabled (adds the disabled class).

```
TsgcHTMLNavItem *item = oNavBar->Items->Add();
item->Text = "Docs";
item->Href = "/docs";
item->Active = false;
```

Output

The inherited **HTML** property returns the complete <nav> block, including the collapse container and the responsive toggler button. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 8 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Sidebar

TsgcHTMLComponent_Sidebar renders a vertical Bootstrap sidebar with an optional brand, a collection of navigation links (with icons, headers and dividers) and an optional footer.

Introduction

TsgcHTMLComponent_Sidebar emits a vertical `<div class="d-flex flex-column">` panel: an optional brand at the top, a stack of `nav-pills` links built from the **Items** collection, and an optional footer. Items can be ordinary links, section *headers* or horizontal *dividers*, and each link may carry an *Icon*. You choose a light or dark scheme with **Dark**, fix the panel to the viewport with **Fixed**, set its **Width**, then read the markup through the inherited **HTML** property.

Each entry is a **TsgcHTMLSidebarItem** exposing *Text*, *Href*, *Icon*, *Active*, *Divider* and *Header*, so a complete menu is described entirely as data.

Quick Start

1. Drop a **TsgcHTMLComponent_Sidebar** component on the form (or create it at runtime).

2. Set the **Brand** and **Width**, then add one item per entry to the **Items** collection.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Sidebar *oSidebar = new TsgcHTMLComponent_Sidebar(NULL);
try
{
  oSidebar->Brand = "Acme";
  oSidebar->Dark = true;
  oSidebar->Width = "250px";
  TsgcHTMLSidebarItem *item;
  item = oSidebar->Items->Add(); item->Text = "Dashboard"; item->Href = "/"; item->Active = true;
  item = oSidebar->Items->Add(); item->Text = "Reports"; item->Href = "/reports";
  item = oSidebar->Items->Add(); item->Divider = true;
  html = oSidebar->HTML;
}
__finally
{
  delete oSidebar;
}
```

Appearance

What it does: *Brand* and *BrandHref* set the heading at the top and its link. *Dark* chooses a dark or light scheme, *Fixed* pins the panel to the left of the viewport, and *Width* sets its width. *CSSClass* appends extra classes and *SidebarID* sets the element id.

Property	Type	Default	Description
Brand	string	''	Heading text shown at the top of the sidebar.

BrandHref	string	'#'	Link target for the brand heading.
Width	string	'250px'	CSS width of the sidebar panel.
Dark	Boolean	True	True uses a dark scheme (bg-dark), False a light one (bg-light).
Fixed	Boolean	True	Pins the panel with position:fixed to the top-left of the viewport.
SidebarID	string	'sgcSide-bar'	Id of the generated sidebar div.
CSSClass	string	''	Extra CSS classes appended to the sidebar div.

Footer

What it does: An optional footer is drawn below a horizontal rule at the bottom of the panel. Supply raw markup with *FooterHTML* for full control, or plain text with *FooterText* for a muted caption. When both are set, *FooterHTML* takes precedence.

Property	Type	Default	Description
FooterHTML	string	''	Raw HTML rendered as the footer (takes precedence over Footer-Text).
FooterText	string	''	Plain text rendered as a muted footer caption.

Items

What it does: The **Items** collection holds the menu entries. Call **Items.Add** to append an entry, then set its sub-properties. Each entry is a **TsgcHTMLSidebarItem** exposing the following published properties.

Item Property	Type	Default	Description
Text	string	''	Caption of the link, or the heading text when Header is True.
Href	string	''	Link target written to the anchor's href attribute.
Icon	string	''	Optional icon markup rendered before the link text.
Active	Boolean	False	Marks the link as the current page (adds the active class).
Divider	Boolean	False	When True the entry renders as a horizontal rule instead of a link.
Header	Boolean	False	When True the entry renders as a muted section heading in-stead of a link.

```
TsgcHTMLSidebarItem *item;
item = oSidebar->Items->Add(); item->Header = true; item->Text = "Admin";
item = oSidebar->Items->Add(); item->Text = "Users"; item->Href = "/users";
```

Output

The inherited **HTML** property returns the complete sidebar `<div>`, including the brand, the link list and any footer. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** alongside your main content area to produce a full HTML document.

Reference

- Properties — 10 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Site

TsgcHTMLComponent_Site builds an entire web page — header, navigation menu, main content and footer — from a single component, with a swappable Layout template.

Introduction

TsgcHTMLComponent_Site is a **whole-page composer**. Where the other HTML components each render one widget, Site assembles a complete document: a header (with brand, user box, theme and language switchers), a navigation menu, the main content area and a footer. It composes the **NavBar** and **Sidebar** components and wraps everything in a **TsgcHTMLTemplate_Bootstrap**, so a full HTML page comes out of the inherited **HTML** property (or **GetHTML** in .NET) with no hand-written markup of your own.

The page shape is chosen with a single **Layout** property (sidebar left or right, top navigation, icon rail, off-canvas, or a top bar combined with a left sidebar). A **Preset** configures Layout and Theme together for a common page type, while the read-only **Brand**, **Theme**, **Header**, **Footer** and **Menu** sub-objects fine-tune every region. You add navigation with **AddMenu** and page body with **AddContent** or **AddSection**.

Quick Start

1. Drop a **TsgcHTMLComponent_Site** component on the form (or create it at runtime).

2. Choose a **Layout**, set the **Theme** and **Brand**, then add menu links and content.

3. Read the **HTML** property (or call **GetHTML**) and serve the finished page.

```
TsgcHTMLComponent_Site *oSite = new TsgcHTMLComponent_Site(NULL);
try
{
  oSite->Layout = slTopNavSidebarLeft;
  oSite->Theme->Preset = stpViolet;
  oSite->Theme->Mode = stmLight;
  oSite->Brand->Text = "Acme";
  oSite->AddMenu("Dashboard", "/", "")->Active = true;
  oSite->AddMenu("Reports", "/reports", "");
  oSite->AddSection("Overview", "<p>Welcome back.</p>", "overview");
  oSite->AddContent("<div class=\"alert alert-info\">All systems normal</div>");
  oSite->Footer->Text = "© 2026 Acme";
  html = oSite->HTML;
}
__finally
{
  delete oSite;
}
```

Layout Templates

What it does: The **Layout** property (type *TsgcHTMLSiteLayout*) selects the overall page shape. The same brand, menu and content are re-arranged into whichever template you choose.

Value	Description

slSidebarLeft	A fixed vertical sidebar on the left with the brand, menu and the main content area to its right.
slSidebarRight	The same as slSidebarLeft, but the sidebar is placed on the right.
slTopNav	A horizontal top navigation bar above a centred or fluid content area, with no sidebar.
slTopNavSide-barLeft	A top bar carrying the brand and header actions, with a left sidebar below it. This is the default.
slIconRail	A narrow icon-only rail fixed on the left (each item shows its icon or first letter), plus a top bar and the content beside it.
slOffcanvas	A top bar with a hamburger toggle that slides the menu in as an off-canvas panel, ide-al for mobile-first apps.

Presets

What it does: The **Preset** property (type *TsgcHTMLSitePreset*) configures *Layout* and *Theme* together for a com-mon page type. Assigning a preset sets several properties at once; you can still override any of them afterwards.

Value	Description
spCustom	No preset applied; you configure Layout and Theme yourself. This is the default.
spAdmin	Top bar with a left sidebar, blue theme, dark sidebar and a visible header.
spDashboard	Icon rail layout, violet theme, dark mode.
spPortal	Top navigation, emerald theme, fixed-width (non-fluid) content.
spDocs	Left sidebar, blue theme, light sidebar.
spLanding	Top navigation, blue theme, fixed-width content and a visible header.
spApp	Off-canvas menu, blue theme.

Theme

What it does: The read-only **Theme** sub-object sets the colour scheme. *Preset* picks an accent palette, *Mode* chooses light, dark or system, *Primary* overrides the accent with an explicit CSS colour, and *SidebarDark* draws the sidebar on a dark background.

Proper-ty	Type	De-fault	Description

Preset	TsgcHTMLSiteThemePreset	stpBlue	Accent palette: stpCustom, stpBlue, stpViolet, stpEmerald, stpSlate or stpDark.
Mode	TsgcHTMLSiteThemeMode	stmLight	stmLight, stmDark or stmSystem (follow the operating system).
Primary	string	''	Explicit CSS colour for the primary accent; overrides the preset when set.
SidebarDark	Boolean	True	Renders the sidebar with a dark background.

Brand

What it does: The read-only **Brand** sub-object sets the identity shown in the header (and in the sidebar for sidebar layouts).

Property	Type	Default	Description
Text	string	''	Brand text shown next to the logo.
LogoURL	string	''	Optional logo image URL rendered before the text.
Href	string	'#'	Link target for the brand.

Header

What it does: The read-only **Header** sub-object controls the top bar: whether it is shown, its optional theme and language switchers, the signed-in user box and a logout link. Use *ExtraHTML* to inject your own controls.

The header follows the request session when the page is rendered by the HTMX engine with an Auth (or when you set the request context with sgcHTMLSetRequestAuth and sgcHTMLSetRequestSession). With *UserName* empty, the user box shows the DisplayName of the session. With *LogoutHref* empty, the logout is a small POST form to Auth.LogoutPath that carries the CSRF token of the session, because a GET link would let any page sign the user out; the built-in logout endpoint only accepts that POST. Without a request auth, or with LogoutHref set, the logout is a plain link. See Sessions, CSRF and Authentication.

Property	Type	Default	Description
Visible	Boolean	True	Shows or hides the header region.
ShowThemeSwitcher	Boolean	False	Adds a button that toggles light and dark mode in the browser.

ShowLan-guageSwitcher	Boolean	False	Adds a language button to the header.
ShowUser	Boolean	False	Shows the signed-in user box (avatar and name).
UserName	string	''	Name shown in the user box; its initial is used when no avatar is set. When empty, the DisplayName of the re-quest session is shown.
UserAvatarURL	string	''	Avatar image URL for the user box.
ShowLogout	Boolean	False	Adds a logout control to the header: a POST form when a request auth is present and LogoutHref is empty, a link otherwise.
LogoutHref	string	''	Target of the logout link. When empty, the POST form goes to Auth.LogoutPath, or the link goes to /logout when there is no request auth.
ExtraHTML	string	''	Raw markup appended to the right side of the header.

Footer

What it does: The read-only **Footer** sub-object draws an optional centred footer below the content. Supply raw markup with *HTML* for full control, or plain text with *Text*. When both are set, *HTML* takes precedence.

Property	Type	Default	Description
Visible	Boolean	True	Shows or hides the footer.
Text	string	''	Plain text rendered as a muted footer caption.
HTML	string	''	Raw HTML rendered as the footer (takes precedence over Text).

Menu

What it does: The read-only **Menu** is a **TsgcHTMLSidebarItems** collection holding the navigation entries. Call **AddMenu** (or **Menu.Add**) to append an entry. Each entry is a **TsgcHTMLSidebarItem** that can be a link, a section *Header* or a *Divider*, and may carry an *Icon*.

Item Property	Type	Default	Description
Text	string	''	Caption of the link, or the heading text when Header is True.
Href	string	''	Link target written to the anchor's href attribute.

Icon	string	''	Optional icon markup rendered before the link text.
Active	Boolean	False	Marks the link as the current page (adds the active class).
Divider	Boolean	False	When True the entry renders as a horizontal rule instead of a link.
Header	Boolean	False	When True the entry renders as a muted section heading instead of a link.

Page Options

What it does: These properties size and label the page. *Title* sets the document title, *Lang* the html language, *SidebarWidth* the width reserved for the sidebar and *ContentFluid* whether the content spans the full width. *CustomCSS*, *CustomHead* and *BodyEndHTML* inject your own CSS, head markup and end-of-body markup.

Property	Type	Default	Description
Title	string	''	Text written to the document <title>.
Lang	string	'en'	Value of the html lang attribute.
SidebarWidth	string	'250px'	CSS width reserved for the sidebar in sidebar layouts.
ContentFluid	Boolean	True	True uses a full-width container; False a centred fixed-width one.
CustomCSS	string	''	Extra CSS appended after the generated theme CSS.
CustomHead	string	''	Raw markup injected into the document <head>.
BodyEndHTML	string	''	Raw markup appended just before the closing </body> tag.

Events

What it does: *OnPrepareTemplate* hands you the page template the Site has just set up, right before the page is rendered, so you can set anything the Site does not expose: the navigation options (Boost, DefaultSwap, LoadingIndicator and the others), extra head nodes or the PWA options. See Navigation and Live Updates.

Event	Type	Description

OnPre-pareTem-plate	TsgcHTMLSitePre-pareTemplateEvent	Fired with the TsgcHTMLTemplate_Bootstrap of the page, after the Site has configured it and before it is rendered.

```
void __fastcall TForm1::SitePrepareTemplate(TObject* Sender,
  TsgcHTMLTemplate_Bootstrap* const aTemplate)
{
  aTemplate->Boost = true;
  aTemplate->LoadingIndicator = true;
}
```

Methods

What it does: Build the page body with these methods. **AddMenu** appends a navigation link and returns it, **AddContent** appends raw markup, **AddSection** appends a titled section with an underlined heading, and **ClearContent** removes everything you added. **GetHTML** returns the finished page.

```
    oSite->AddMenu("Users", "/users", "👤");
    oSite->AddSection("Team", oList->HTML, "team");
    oSite->AddContent("<div class=\"card\">...</div>");
```

Output

The inherited **HTML** property (or **GetHTML** in .NET) returns a complete HTML document: the Bootstrap template with the generated theme CSS, the chosen layout wrapping the header, menu, every section and raw block you added, and the optional footer. Serve this string directly from your web server as the page body.

Reference

- Properties — 14 properties available on the component.
- Methods — 5 public methods.
- Events — 1 published event, fired with the page template before the page is rendered.

TsgcHTMLComponent_Breadcrumb

TsgcHTMLComponent_Breadcrumb renders a Bootstrap breadcrumb trail from a collection of items, with an optional custom divider.

Introduction

TsgcHTMLComponent_Breadcrumb emits a Bootstrap `<nav aria-label="breadcrumb">` containing an ordered list of `breadcrumb-item` entries built from the **Items** collection. Each item is a link except the last one, which is rendered as the current page. You can change the separator with the **Divider** property, then read the finished markup through the inherited **HTML** property.

Each entry is a **TsgcHTMLBreadcrumbItem** exposing *Text*, *Href* and *Active*. The component automatically marks the final item (or any item with *Active* set) as the current page, so you describe the trail purely as data.

Quick Start

1. Drop a **TsgcHTMLComponent_Breadcrumb** component on the form (or create it at runtime).

2. Add one item per level to the **Items** collection, from the root to the current page.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Breadcrumb *oBreadcrumb = new TsgcHTMLComponent_Breadcrumb(NULL);
try
{
  TsgcHTMLBreadcrumbItem *item;
  item = oBreadcrumb->Items->Add(); item->Text = "Home"; item->Href = "/";
  item = oBreadcrumb->Items->Add(); item->Text = "Library"; item->Href = "/library";
  item = oBreadcrumb->Items->Add(); item->Text = "Data";
  html = oBreadcrumb->HTML;
}
__finally
{
  delete oBreadcrumb;
}
```

Appearance

What it does: *Divider* overrides the default Bootstrap separator between items by setting the `--bs-breadcrumb-divider` CSS variable. *BreadcrumbID* assigns an id to the generated nav element.

Property	Type	Default	Description
Divider	string	''	Custom separator character or symbol; empty keeps the Bootstrap default.

Bread-crumbID	string	''	Optional id for the generated nav element; omitted when empty.

Items

What it does: The **Items** collection holds the trail entries from root to current page. Call **Items.Add** to append a level, then set its sub-properties. Each entry is a **TsgcHTMLBreadcrumbItem** exposing the following published properties. The last entry (or any entry with *Active* set) is rendered as the current page rather than a link.

Item Property	Type	Default	Description
Text	string	''	Visible caption of the level (HTML-encoded on output).
Href	string	''	Link target; ignored for the current-page entry.
Active	Boolean	False	Forces this entry to render as the current page; the last entry is always current.

```
oBreadcrumb->Divider = ">";
TsgcHTMLBreadcrumbItem *item;
item = oBreadcrumb->Items->Add(); item->Text = "Home"; item->Href = "/";
item = oBreadcrumb->Items->Add(); item->Text = "Profile";
```

Output

The inherited **HTML** property returns the complete <nav> with its ordered list. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 3 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Tabs

TsgcHTMLComponent_Tabs renders a Bootstrap tabbed interface (tabs, pills or underline) from a collection of tab items, each with its own title and content panel.

Introduction

TsgcHTMLComponent_Tabs emits a Bootstrap tab navigation `<ul>` plus a matching `tab-content` region. Each tab is a **TsgcHTMLTabItem** with its own *Title* button and *Content* panel, and the component wires the `data-bs-toggle="tab"` attributes so Bootstrap can switch panels in the browser. You pick the visual **Style**, optionally lay the tabs out vertically, then read the finished markup through the inherited **HTML** property.

Tabs render in the order they are added; mark exactly one as *Active* to choose the panel shown first, and use *Disabled* to grey out a tab.

Quick Start

1. Drop a **TsgcHTMLComponent_Tabs** component on the form (or create it at runtime).

2. Add one item per tab to the **Items** collection, set each tab's **Title** and **Content**, and mark one **Active**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Tabs *oTabs = new TsgcHTMLComponent_Tabs(NULL);
try
{
  oTabs->Style = tsTab;
  TsgcHTMLTabItem *item;
  item = oTabs->Items->Add(); item->Title = "Overview"; item->Content = "<p>Welcome</p>"; item->Active = true
  item = oTabs->Items->Add(); item->Title = "Details"; item->Content = "<p>More</p>";
  html = oTabs->HTML;
}
__finally
{
  delete oTabs;
}
```

Layout and Style

What it does: *Style* selects the tab look (classic tabs, pills or underline). *Vertical* stacks the tabs down the left with the panels beside them. *Fill* and *Justify* spread the tab buttons across the available width. *TabsID* sets the base id from which each panel's id is derived.

Property	Type	Default	Description
Style	TsgcHTMLTabStyle	tsTab	Tab look: tsTab (nav-tabs), tsPill (nav-pills) or tsUnderline (nav-underline).

Vertical	Boolean	False	Lays the tabs out vertically with content beside them.
Fill	Boolean	False	Spreads tab buttons to fill the available width (nav-fill).
Justify	Boolean	False	Gives all tab buttons equal width (nav-justified).
TabsID	string	'sgcTab-s'	Base id used to build each tab and panel id.

Items

What it does: The **Items** collection holds the tabs. Call **Items.Add** to append a tab, then set its sub-properties. Each tab is a **TsgcHTMLTabItem** exposing the following published properties.

Item Property	Type	Default	Description
Title	string	''	Caption shown on the tab button.
Content	string	''	HTML markup rendered inside the tab's content panel.
Active	Boolean	False	Marks this tab as the one selected first (set on exactly one item).
Disabled	Boolean	False	Renders the tab as disabled (adds the disabled class).
TabID	string	''	Optional explicit id for the tab and its panel; auto-generated when empty.

```
TsgcHTMLTabItem *item = oTabs->Items->Add();
item->Title = "Settings";
item->Content = "<form>...</form>";
item->TabID = "settings";
```

Output

The inherited **HTML** property returns the tab navigation list and the content region, with the Bootstrap toggle attributes already in place. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document. Bootstrap's bundled JavaScript switches the panels at runtime.

Reference

- Properties — 6 properties available on the component.

- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Pagination

TsgcHTMLComponent_Pagination renders a Bootstrap pagination control, a numbered list of page links with optional Previous/Next and First/Last buttons, sized and aligned to taste.

Introduction

TsgcHTMLComponent_Pagination emits a `<nav>` containing a Bootstrap `<ul class="pagination">`. You tell it the current page and either the total number of pages, or the total number of items and the page size (from which the page count is derived). The component then draws a window of page numbers centred on the current page, marks the current page active, and links each page through a base URL. The finished markup is returned by the inherited **HTML** property.

The window of visible page numbers is controlled by **MaxVisible**. When the control has only one page (or fewer) it renders nothing, so it is safe to place on every list page unconditionally.

Quick Start

1. Drop a **TsgcHTMLComponent_Pagination** component on the form (or create it at runtime).

2. Set **TotalItems** and **PageSize** (or **TotalPages** directly), **CurrentPage** and a **BaseURL**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Pagination *oPager = new TsgcHTMLComponent_Pagination(NULL);
try
{
  oPager->TotalItems = 137;
  oPager->PageSize = 10;
  oPager->CurrentPage = 4;
  oPager->BaseURL = "?page=";
  oPager->ShowFirstLast = true;
  html = oPager->HTML;
}
__finally
{
  delete oPager;
}
```

Paging Model

What it does: *CurrentPage* marks the active page. The page count comes from *TotalPages*, or is computed from *TotalItems* and *PageSize* when both are set (so you can pass a record count straight from a query). *BaseURL* is prefixed to each page number to build the link, and *PaginationID* sets the id of the generated `<nav>`.

Property	Type	Default	Description

CurrentPage	Integer	1	The page currently selected; it is rendered as the active item.
TotalPages	Integer	1	Total number of pages. Recomputed from TotalItems / PageSize when those are set.
TotalItems	Integer	0	Total record count. When greater than zero the page count is derived from it.
PageSize	Integer	10	Number of items per page, used to derive TotalPages from TotalItems.
BaseURL	string	'?page='	Prefix prepended to each page number to form the link href.
PaginationID	string	''	Optional id attribute for the generated <nav> element.

Appearance and Navigation Buttons

What it does: *Size* and *Align* control the Bootstrap sizing and horizontal alignment of the list. *MaxVisible* limits how many numbered pages are drawn around the current page. *ShowPrevNext* adds Previous/Next buttons and *ShowFirstLast* adds First/Last buttons.

Property	Type	Default	Description
Size	TsgcHTMLPaginationSize	psDefault	List size: psDefault, psSmall or psLarge.
Align	TsgcHTMLPaginationAlign	paCenter	Horizontal alignment: paStart, paCenter or paEnd.
MaxVisible	Integer	5	Maximum number of numbered page links shown around the current page.
ShowPrevNext	Boolean	True	Shows the Previous and Next buttons.
ShowFirstLast	Boolean	False	Shows the First and Last buttons.

```
oPager->Size = psSmall;
oPager->Align = paEnd;
oPager->MaxVisible = 7;
```

Output

The inherited **HTML** property returns a `<nav>` wrapping a Bootstrap `<ul class="pagination">`. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 11 properties available on the component.
- Methods — this component has no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_Toolbar

TsgcHTMLComponent_Toolbar renders a Bootstrap button toolbar, a row of buttons or links grouped together, with optional icons, tooltips and separators.

Introduction

TsgcHTMLComponent_Toolbar emits a `<div class="btn-toolbar">` containing one or more Bootstrap button groups. Each entry in the **Items** collection is a button (or, when it has an Href, a link styled as a button). A separator item starts a new button group and inserts a small gap, so you can visually cluster related actions. The finished markup is returned by the inherited **HTML** property.

You can build the toolbar at design time by adding **TsgcHTMLToolbarItem** entries to the **Items** collection, or at runtime with the convenience methods **AddButton** and **AddSeparator**.

Quick Start

1. Drop a **TsgcHTMLComponent_Toolbar** component on the form (or create it at runtime).

2. Add buttons with **AddButton**, optionally calling **AddSeparator** between groups.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Toolbar *oToolbar = new TsgcHTMLComponent_Toolbar(NULL);
try
{
  oToolbar->AddButton("New", bsPrimary);
  oToolbar->AddButton("Edit", bsOutlinePrimary);
  oToolbar->AddSeparator();
  oToolbar->AddButton("Delete", bsOutlineDanger);
  html = oToolbar->HTML;
}
__finally
{
  delete oToolbar;
}
```

Items

What it does: The toolbar is built from the **Items** collection of **TsgcHTMLToolbarItem**. Each item exposes *Text*, *Href*, *ButtonStyle*, *Icon*, *Disabled*, *Separator* and *Tooltip*. An item with an Href is rendered as a link styled as a button; otherwise it is a `<button>`. Marking an item as a separator closes the current button group and inserts a gap.

```
TsgcHTMLToolbarItem *oItem = oToolbar->Items->Add();
oItem->Text = "Refresh";
oItem->ButtonStyle = bsOutlineSecondary;
oItem->Icon = "↻";
oItem->Tooltip = "Reload the list";
```

Sizing

What it does: *Size* applies a Bootstrap button-group size to every group in the toolbar, and *ToolbarID* sets the id of the wrapping `<div>`.

Property	Type	Default	Description
Items	TsgcHTMLToolbarItems		Collection of TsgcHTMLToolbarItem buttons, links and separators.
Size	TsgcHTMLToolbarSize	tbsDefault	Button-group size: tbsDefault, tbsSmall or tbsLarge.
ToolbarID	string	''	Optional id attribute for the generated .

Output

The inherited **HTML** property returns a `<div class="btn-toolbar">` containing one or more `<div class="btn-group">` blocks. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 3 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_TreeView

TsgcHTMLComponent_TreeView renders a collapsible tree of nodes as a Bootstrap list group, with optional connector lines, per-level indentation and custom expand, collapse and leaf icons.

Introduction

TsgcHTMLComponent_TreeView emits a `<div class="list-group">` in which each node is a row indented by its depth. Nodes with children get a toggle that collapses or expands the branch using Bootstrap's collapse behaviour. You build the tree by adding **TsgcHTMLTreeNode** items to the **Nodes** collection (each node carries a simple list of child captions), or load it from a dataset with **LoadFromDataSet**. The finished markup is returned by the inherited **HTML** property.

Connector lines are drawn with a small CSS rule emitted alongside the markup; their color is taken from **LineColor** when set, or from the Bootstrap palette through **LineColorStyle**.

Quick Start

1. Drop a **TsgcHTMLComponent_TreeView** component on the form (or create it at runtime).

2. Add nodes to the **Nodes** collection, or call **LoadFromDataSet** to build the tree from a query.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_TreeView *oTree = new TsgcHTMLComponent_TreeView(NULL);
try
{
  TsgcHTMLTreeNode *oNode = oTree->Nodes->Add();
  oNode->Text = "Documents";
  oNode->Expanded = true;
  oNode->Children->Add("Invoices");
  oNode->Children->Add("Reports");
  html = oTree->HTML;
}
__finally
{
  delete oTree;
}
```

Nodes

What it does: The tree is built from the **Nodes** collection of **TsgcHTMLTreeNode**. Each node exposes *Text*, *Href*, *Icon*, *Expanded*, *Selected*, *NodeID* and a *Children* string list of child captions. A node with children renders a collapse toggle; the *Expanded* flag decides whether the branch starts open. Bind a whole tree from data with **LoadFromDataSet**.

```
// Build the tree from an adjacency-list query (id, parent_id, name)
oTree->LoadFromDataSet(qryFolders, "id", "parent_id", "name");
```

Appearance

What it does: *ShowLines* draws connector lines down the tree, colored by *LineColor* or *LineColorStyle*. *IndentSize* sets the per-level indentation in pixels. *ExpandedIcon*, *CollapsedIcon* and *LeafIcon* override the default toggle and bullet glyphs. *Selectable* and *TreeID* control selection styling and the wrapper id.

Property	Type	Default	Description
Nodes	TsgcHTMLTreeNodes		Collection of TsgcHTMLTreeNode items drawn in the tree.
TreeID	string	'sgcTreeView'	The id of the wrapping .
ShowLines	Boolean	True	Draws connector lines down the left edge of the tree.
Selectable	Boolean	False	Enables selection styling for nodes marked Selected.
LineColor	string	''	Explicit CSS color for the connector lines. Overrides LineColorStyle.
LineColorStyle	TsgcHTMLColor	hcLight	Bootstrap palette color used for the lines when LineColor is empty.
IndentSize	Integer	20	Indentation in pixels added for each tree level.
ExpandedIcon	string	▼	Glyph shown on an expanded parent node.
CollapsedIcon	string	▶	Glyph shown on a collapsed parent node.
LeafIcon	string	•	Glyph shown on a node with no children.

Output

The inherited **HTML** property returns a `<div class="list-group sgc-tree">` with one row per node and a nested collapse block per branch. A small CSS rule for the connector lines is emitted with it. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 10 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Stepper

TsgcHTMLComponent_Stepper renders a multi-step progress indicator, a horizontal or vertical row of numbered steps marked upcoming, current or completed, with optional per-step content.

Introduction

TsgcHTMLComponent_Stepper emits a stepper (also called a wizard progress bar): a sequence of circles joined by connectors, each carrying a title and a short description. Every step has a state (*upcoming*, *current* or *completed*) that drives its circle color and the connector leading up to it. The component lays the steps out horizontally or vertically and can render the content panel of the current step beneath the indicator. The finished markup, along with the CSS it needs, is returned by the inherited **HTML** property.

You build the steps by adding **TsgcHTMLStepItem** entries to the **Items** collection. Colors can be set either as explicit CSS values or by picking from the Bootstrap palette through the matching *...Style* property.

Quick Start

1. Drop a **TsgcHTMLComponent_Stepper** component on the form (or create it at runtime).

2. Add steps to the **Items** collection and set each step's **State**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Stepper *oStepper = new TsgcHTMLComponent_Stepper(NULL);
try
{
  TsgcHTMLStepItem *oStep;
  oStep = oStepper->Items->Add(); oStep->Title = "Account"; oStep->State = ssCompleted;
  oStep = oStepper->Items->Add(); oStep->Title = "Profile"; oStep->State = ssCurrent;
  oStep = oStepper->Items->Add(); oStep->Title = "Confirm"; oStep->State = ssUpcoming;
  html = oStepper->HTML;
}
__finally
{
  delete oStepper;
}
```

Steps and Layout

What it does: The stepper is built from the **Items** collection of **TsgcHTMLStepItem**. Each step exposes *Title*, *Description*, *Content*, *State* and *Icon*. *Layout* chooses horizontal or vertical, and *ShowContent* decides whether the current step's content panel is rendered.

Property	Type	Default	Description
Items	TsgcHTML-StepItems		Collection of TsgcHTMLStepItem steps drawn by the component.

Layout	TsgcHTMLStepper-Layout	slHorizon-tal	Step layout: slHorizontal or slVertical.
ShowCon-tent	Boolean	True	Renders the content panel for the current (or each) step.
StepperID	string	'sgcStep-per'	The id of the generated stepper container.

State Colors

What it does: Each state has a circle color (and the upcoming state also has a text color), set either as an explicit CSS value (*CompletedColor*, *CurrentColor*, *UpcomingColor*, *UpcomingTextColor*, *ConnectorColor*) or by picking from the Bootstrap palette through the matching *...Style* property. When the explicit color is empty the *...Style* value is used.

Property	Type	Default	Description
CompletedColor	string	''	Explicit circle color for completed steps. Overrides CompletedColorStyle.
CompletedCol-orStyle	TsgcHTML-Color	hcSuccess	Palette color for completed steps when Completed-Color is empty.
CurrentColor	string	''	Explicit circle color for the current step. Overrides CurrentColorStyle.
CurrentCol-orStyle	TsgcHTML-Color	hcPrimary	Palette color for the current step when CurrentColor is empty.
UpcomingColor	string	''	Explicit circle color for upcoming steps. Overrides Up-comingColorStyle.
UpcomingCol-orStyle	TsgcHTML-Color	hcLight	Palette color for upcoming steps when UpcomingCol-or is empty.
UpcomingText-Color	string	''	Explicit text color for upcoming steps. Overrides Up-comingTextColorStyle.
UpcomingText-ColorStyle	TsgcHTML-Color	hcSec-ondary	Palette text color for upcoming steps when Upcom-ingTextColor is empty.
ConnectorColor	string	'#dee2e6'	Explicit color for connectors between not-yet-com-pleted steps. Overrides ConnectorColorStyle.
ConnectorCol-orStyle	TsgcHTML-Color	hcLight	Palette color for connectors when ConnectorColor is empty.

Sizing and Glyphs

What it does: *CircleSize* and *ConnectorHeight* size the circles and connectors, *StepFontSize* and *StepDescFont-Size* size the title and description text, and *CompletedIcon* is the glyph drawn inside a completed step's circle.

Property	Type	Default	Description
CircleSize	Integer	40	Diameter of the step circle in pixels (horizontal layout).
ConnectorHeight	Integer	2	Thickness of the connector line in pixels.
CompletedIcon	string	✓	Glyph drawn inside the circle of a completed step.
StepFontSize	string	'0.9rem'	CSS font size for the step title.
StepDescFontSize	string	'0.8rem'	CSS font size for the step description.

Output

The inherited **HTML** property returns the stepper markup along with the small CSS block it needs. Horizontal layout produces a flex row of `sgc-step` circles; vertical layout produces a stacked `sgc-stepper-v` list. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 19 properties available on the component.
- Methods — this component has no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_Dropdown

TsgcHTMLComponent_Dropdown renders a Bootstrap dropdown button with a menu built from a collection of items (links, headers and dividers).

Introduction

TsgcHTMLComponent_Dropdown emits a Bootstrap `<div class="dropdown">` containing a toggle `<button>` and a `<ul class="dropdown-menu">` list. You set the button caption with **ButtonText**, pick a button color with **ButtonStyleEnum**, then add one entry per menu line to the **Items** collection. The component builds the markup for you and returns it through the inherited **HTML** property.

Each entry is a **TsgcHTMLDropdownItem** that can be a normal link, a non-clickable *Header*, or a *Divider*, so the whole menu is described as data and no hand-written HTML is needed.

Quick Start

1. Drop a **TsgcHTMLComponent_Dropdown** component on the form (or create it at runtime).

2. Set the **ButtonText**, pick a **ButtonStyleEnum**, then add one item per menu line to the **Items** collection.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Dropdown *oDropdown = new TsgcHTMLComponent_Dropdown(NULL);
try
{
  oDropdown->ButtonText = "Actions";
  oDropdown->ButtonStyleEnum = bsPrimary;
  TsgcHTMLDropdownItem *item;
  item = oDropdown->Items->Add(); item->Text = "Edit"; item->Href = "/edit";
  item = oDropdown->Items->Add(); item->Text = "Duplicate"; item->Href = "/copy";
  item = oDropdown->Items->Add(); item->Divider = true;
  item = oDropdown->Items->Add(); item->Text = "Delete"; item->Href = "/delete";
  html = oDropdown->HTML;
}
__finally
{
  delete oDropdown;
}
```

Button

What it does: *ButtonText* sets the toggle caption. *ButtonStyleEnum* chooses the Bootstrap button color, and *ButtonClass* overrides it with your own classes when you need full control. *Split* selects a split-button layout, and *DropdownID* sets the id of the generated wrapper.

Property	Type	Default	Description
ButtonText	string	'Dropdown'	Caption of the toggle button.

Button-StyleEnum	TsgcHTMLBut-tonStyle	bsSec-ondary	Bootstrap button color used when ButtonClass is empty.
ButtonClass	string	''	Explicit button classes; when set, overrides But-tonStyleEnum.
Split	Boolean	False	Selects a split-button dropdown layout.
DropdownID	string	'sgcDrop-down'	The id of the generated dropdown wrapper ele-ment.

Items

What it does: The **Items** collection holds the menu entries. Call **Items.Add** to append an entry, then set its sub-properties. Each item is a **TsgcHTMLDropdownItem** that is rendered as a link, or as a header or divider when the matching flag is set.

Item Property	Type	Default	Description
Text	string	''	Visible caption of the entry (HTML-encoded on output).
Href	string	''	Link target written to the entry's href attribute.
Divider	Boolean	False	Renders a horizontal separator instead of a link.
Header	Boolean	False	Renders a non-clickable section header instead of a link.
Disabled	Boolean	False	Renders the link as disabled (adds the disabled class).

```
TsgcHTMLDropdownItem *item;
item = oDropdown->Items->Add(); item->Header = true; item->Text = "Account";
item = oDropdown->Items->Add(); item->Text = "Profile"; item->Href = "/profile";
item = oDropdown->Items->Add(); item->Text = "Sign out"; item->Href = "/logout"; item->Disabled = true;
```

Output

The inherited **HTML** property returns the complete `<div class="dropdown">` block, including the toggle button and the menu list. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to pro-duce a full HTML document.

Reference

- Properties — 6 properties available on the component.

- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_ButtonGroup

TsgcHTMLComponent_ButtonGroup renders a Bootstrap button group, a horizontal or vertical row of buttons built from a collection of items.

Introduction

TsgcHTMLComponent_ButtonGroup emits a Bootstrap `<div class="btn-group">` (or `btn-group-vertical`) containing one button per entry in the **Items** collection. You choose the group **Size**, switch between a horizontal and a **Vertical** layout, then add the buttons. The component builds the markup for you and returns it through the inherited **HTML** property.

Each entry is a **TsgcHTMLButtonItem** with its own caption, color, active and disabled flags. When an item has an *Href* it renders as an anchor, otherwise as a real `<button>`, so the group is described entirely as data and no hand-written HTML is needed.

Quick Start

1. Drop a **TsgcHTMLComponent_ButtonGroup** component on the form (or create it at runtime).

2. Pick a **Size** and orientation, then add one item per button to the **Items** collection.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_ButtonGroup *oGroup = new TsgcHTMLComponent_ButtonGroup(NULL);
try
{
  oGroup->Size = bgsLarge;
  TsgcHTMLButtonItem *item;
  item = oGroup->Items->Add(); item->Text = "Left"; item->Active = true;
  item = oGroup->Items->Add(); item->Text = "Middle";
  item = oGroup->Items->Add(); item->Text = "Right";
  html = oGroup->HTML;
}
__finally
{
  delete oGroup;
}
```

Layout

What it does: *Size* selects a small, default or large button group. *Vertical* stacks the buttons instead of placing them in a row. *AriaLabel* sets the accessible name of the group, and *GroupID* sets the id of the generated wrapper.

Property	Type	Default	Description
Size	TsgcHTMLButton-GroupSize	bgsDefault	Button group size: bgsDefault, bgsSmall or bgsLarge.

Vertical	Boolean	False	Stacks the buttons vertically instead of in a row.
AriaLabel	string	'Button group'	Accessible name written to the group's aria-label attribute.
GroupID	string	''	Id of the generated wrapper element; omitted when empty.

Items

What it does: The Text of an item is plain text and is HTML-encoded, so markup in a caption is shown literally, and its Href is sanitized and attribute-encoded. The **Items** collection holds the buttons. Call **Items.Add** to append a button, then set its sub-properties. Each item is a **TsgcHTMLButtonItem** exposing the following published properties.

Item Property	Type	Default	Description
Text	string	''	Visible caption of the button.
Href	string	''	When set, the button renders as an anchor pointing here; otherwise as a button element.
BtnClass	string	''	Explicit button classes; when set, overrides ButtonStyle.
Active	Boolean	False	Marks the button as pressed (adds the active class).
Disabled	Boolean	False	Renders the button as disabled (adds the disabled class).
ButtonStyle	TsgcHTMLButtonStyle	bsPrimary	Bootstrap button color used when BtnClass is empty.

```
TsgcHTMLButtonItem *item;
item = oGroup->Items->Add(); item->Text = "Save"; item->ButtonStyle = bsSuccess;
item = oGroup->Items->Add(); item->Text = "Cancel"; item->ButtonStyle = bsOutlineSecondary;
```

Output

The inherited **HTML** property returns the complete `<div class="btn-group">` block with one button or anchor per item. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 5 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_ListGroup

TsgcHTMLComponent_ListGroup renders a Bootstrap list group, a flexible list of items with optional links, badges and secondary text, built from a collection or loaded from a dataset.

Introduction

TsgcHTMLComponent_ListGroup emits a Bootstrap list group as a `<ul>`, `<ol>` or `<div>` depending on the options you set, with one `list-group-item` per entry in the **Items** collection. You add entries with the **AddItem** method, bind them to a dataset with **LoadFromDataSet**, then read the finished markup through the inherited **HTML** property.

Each entry is a **TsgcHTMLListGroupItem** with its own caption, optional *Href*, a contextual *Badge*, an *Active* or *Disabled* flag and optional secondary *Content*, so the whole list is described as data and no hand-written HTML is needed.

Quick Start

1. Drop a **TsgcHTMLComponent_ListGroup** component on the form (or create it at runtime).

2. Add entries with **AddItem** (or load them with **LoadFromDataSet**) and set the list options.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_ListGroup *oList = new TsgcHTMLComponent_ListGroup(NULL);
try
{
  oList->Numbered = true;
  oList->AddItem("Inbox", "/inbox", "12", bgPrimary);
  oList->AddItem("Sent", "/sent");
  oList->AddItem("Drafts", "/drafts", "3", bgSecondary);
  html = oList->HTML;
}
__finally
{
  delete oList;
}
```

List Options

What it does: *Flush* removes the outer borders and rounded corners so the list sits flush inside a card. *Numbered* renders an ordered list with automatic numbering. *Horizontal* lays the items out in a row instead of a column. *ListGroupID* sets the id of the generated list element.

Property	Type	Default	Description
Flush	Boolean	False	Removes borders and rounded corners (adds list-group-flush).
Numbered	Boolean	False	Renders an ordered, numbered list (adds list-group-numbered).

Horizontal	Boolean	False	Lays the items out horizontally (adds list-group-horizontal).
ListGroupID	string	''	Id of the generated list element; omitted when empty.

Items

What it does: The **Items** collection holds the list entries. Add them with **AddItem**, fill them from a dataset with **LoadFromDataSet**, or call **Items.Add** for full control. Each item is a **TsgcHTMLListGroupItem** exposing the following published properties.

Item Property	Type	Default	Description
Text	string	''	Visible caption of the entry.
Href	string	''	When set, the entry renders as a clickable list-group-item-action anchor.
Badge	string	''	Optional pill badge shown at the end of the entry.
BadgeStyle	TsgcHTML-BadgeStyle	bgPrima-ry	Contextual color of the badge.
Active	Boolean	False	Marks the entry as the current selection (adds the active class).
Disabled	Boolean	False	Renders the entry as disabled (adds the disabled class).
Color	TsgcHTML-Color	hcLight	Contextual color variant of the entry.
Content	string	''	Optional secondary text shown under the caption.

```
// Bind a query: a caption column, an optional link column and a badge column
oList->LoadFromDataSet(qryTasks, "Title", "Url", "Count");
```

Output

The inherited **HTML** property returns the complete list group block. When the first item has an `Href` the list is rendered as a `<div>` of anchors; when **Numbered** is set it is an `<ol>`; otherwise it is a `<ul>`. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 5 properties available on the component.
- Methods — 3 public methods.
- Events — this component has no published events.

TsgcHTMLDashboardLayout

TsgcHTMLDashboardLayout is a layout shell that pairs a fixed sidebar with a scrolling main area, ready to host the content sections of a dashboard or admin page.

Introduction

TsgcHTMLDashboardLayout is not a single widget but a **layout shell**. It emits a flex container with a fixed **Sidebar** on the left and a padded main region on the right, then fills that region with the content sections you add. You configure the read-only **Sidebar** sub-object, add sections with **AddSection** or drop in raw markup with **AddRawContent**, and read the finished page body through the inherited **HTML** property.

An optional **DarkMode** injects a complete dark colour theme for the common Bootstrap blocks, so the same content works in both light and dark dashboards without any extra CSS of your own.

Quick Start

1. Drop a **TsgcHTMLDashboardLayout** component on the form (or create it at runtime).

2. Configure the **Sidebar** sub-object, then add one or more content sections with **AddSection**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLDashboardLayout *oLayout = new TsgcHTMLDashboardLayout(NULL);
try
{
  oLayout->Sidebar->Brand = "Admin";
  TsgcHTMLSidebarItem *item;
  item = oLayout->Sidebar->Items->Add(); item->Text = "Overview"; item->Href = "/"; item->Active = true;
  item = oLayout->Sidebar->Items->Add(); item->Text = "Users"; item->Href = "/users";
  oLayout->AddSection("Overview", "<p>Welcome back.</p>", "overview");
  oLayout->FooterText = "© 2026 Acme";
  html = oLayout->HTML;
}
__finally
{
  delete oLayout;
}
```

Sidebar and Footer

What it does: The read-only **Sidebar** sub-object is a full **TsgcHTMLComponent_Sidebar**: set its *Brand*, add navigation *Items*, choose its *Width* and dark or light theme. The layout reserves that width on the left and offsets the main area to match. *FooterText* draws an optional centred footer below the content.

Property	Type	Default	Description

Sidebar	TsgcHTMLComponent_Sidebar	(read-only)	The sidebar rendered on the left. Configure its sub-properties; the layout reserves its width.
Footer-Text	string	''	Optional centred footer text shown below the content. No footer is drawn when empty.
LayoutID	string	'sgc-Dashboard'	Id of the outer layout container.

Main Area and Sections

What it does: *Fluid* and *DarkMode* set the overall feel. *MainPadding* and *MainMinHeight* size the scrolling main region. Each section you add with **AddSection** gets a heading with a coloured bottom border and the margin set by *SectionMarginBottom*; the border colour comes from *SectionTitleBorderColor* (an explicit value) or, when that is empty, from *SectionTitleBorderColorStyle*.

Property	Type	Default	Description
Fluid	Boolean	True	Uses a full-width fluid layout.
DarkMode	Boolean	False	Injects a dark colour theme for the common Bootstrap blocks.
MainPadding	string	'24px'	CSS padding applied to the main content region.
MainMinHeight	string	'100vh'	CSS minimum height of the main content region.
SectionMarginBottom	string	'2rem'	CSS bottom margin between sections added with AddSection.
SectionTitleBorderColor	string	''	Explicit CSS color for the section heading underline; overrides the enum below.
SectionTitleBorderColorStyle	TsgcHTMLColor	hcLight	Contextual color for the section heading underline when SectionTitleBorderColor is empty.

```
oLayout->SectionTitleBorderColorStyle = hcPrimary;
oLayout->AddSection("Sales", oChart->HTML, "sales");
oLayout->AddSection("Recent activity", oList->HTML);
oLayout->AddRawContent("<div class=\"alert alert-info\">All systems normal</div>");
```

Output

The inherited **HTML** property returns the complete dashboard body: the optional dark-mode style block, a flex container holding the fixed sidebar and the offset main region, every section and raw block you added, and the optional footer. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 10 properties available on the component.
- Methods — 3 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Grid

TsgcHTMLComponent_Grid renders a Bootstrap HTML table from columns and rows you supply directly or load from a dataset, with optional client-side sorting, filtering, CSV/PDF export, server-side XLSX export, inline editing, grouping, hierarchical tree data, expandable master-detail rows, saved views, virtual scrolling, client or server paging, frozen and resizable columns, multi-column sort, row selection with bulk actions, typed and validated columns, summaries, conditional formatting, cell renderers, keyboard navigation, live row updates and an AI query bar.

Introduction

TsgcHTMLComponent_Grid emits a Bootstrap `<table>` wrapped, when **Responsive** is True, in a `table-responsive` `<div>`. You define the visible columns through the **Columns** collection (or let **LoadFromDataSet** build them from a dataset), add rows with **AddRow**, then read the inherited **HTML** property to get the complete markup.

Sorting, filtering, export, inline editing and column reordering are produced as a small inline JavaScript block, so they work in the browser without a round trip to the server. When **AIQueryEnabled** is True the grid also renders a natural-language query bar that posts to the server, where the **OnAIQuery** event lets you answer the request, or an assigned **AIAssistant** answers it for you.

Quick Start

1. Drop a **TsgcHTMLComponent_Grid** component on the form (or create it at runtime).

2. Add one or more columns to the **Columns** collection, then add rows with **AddRow** (or call **LoadFromDataSet**).

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Grid *oGrid = new TsgcHTMLComponent_Grid(NULL);
try
{
  oGrid->Columns->Add()->Title = "Name";
  oGrid->Columns->Add()->Title = "City";
  oGrid->AddRow(OPENARRAY(String, ("Alice", "London")));
  oGrid->AddRow(OPENARRAY(String, ("Bob", "Paris")));
  oGrid->ShowSort = true;
  oGrid->ShowFilter = true;
  html = oGrid->HTML;
}
__finally
{
  delete oGrid;
}
```

Columns and Rows

What it does: *Columns* is a collection of **TsgcHTMLGridColumn** items (each with *Name*, *Title*, *Width* and *Align*) that defines the header and the cell order. *Rows* holds the row data as a TStringList, one tab-delimited line per row; use **AddRow** to append a row and **Clear** to empty the rows. Call **LoadFromDataSet** to build both columns and rows from a TDataSet.

Property	Type	Default	Description
Columns	TsgcHTMLGrid-Columns		Collection of TsgcHTMLGridColumn items defining the grid columns.
Rows	TStringList		Row data, one tab-delimited line per row.
TableID	string	”	The id attribute of the generated <table> element.
CSSClass	string	”	Extra CSS class appended to the table class list.

```
// Build columns and rows from a query
oGrid->LoadFromDataSet(qryCustomers);
```

Appearance

What it does: The Bootstrap table styling flags *Striped*, *Bordered*, *Hover*, *Dark* and *Responsive* control how the table looks. They map to the Bootstrap `table-striped`, `table-bordered`, `table-hover` and `table-dark` classes.

Property	Type	Default	Description
Striped	Boolean	True	Adds zebra striping to the rows (table-striped).
Bordered	Boolean	False	Draws borders around all cells (table-bordered).
Hover	Boolean	True	Highlights the row under the pointer (table-hover).
Dark	Boolean	False	Uses the dark table variant (table-dark).
Responsive	Boolean	True	Wraps the table in a table-responsive container.

Interactive Features

What it does: Turn on *ShowSort*, *ShowFilter*, *ShowExport*, *InlineEdit* and *ColumnReorder* to add client-side behaviour through an inline script. *ExportFileName* names the downloaded CSV, *EditMode* chooses cell or row editing, and the grouping (*ShowGrouping*, *GroupByColumn*) and virtual-scroll (*VirtualScroll*, *VirtualScrollURL*, *VisibleRows*) options handle larger datasets.

The client-side CSV export now quotes fields per RFC 4180 (a field that contains a comma, a quote or a line break is quoted and its embedded quotes are doubled), which changes the output for anyone who parsed the previous unquoted CSV, and the client-side numeric sort is now locale-safe: it recognises both ‘.’ and ‘,’ as the decimal separator, so numbers sort identically on every machine.

Property	Type	Default	Description
ShowSort	Boolean	False	Makes column headers clickable to sort rows.
ShowFilter	Boolean	False	Adds a search box that filters rows as you type.
ShowExport	Boolean	False	Adds Export CSV and Export PDF buttons.
ExportFile-Name	string	'export.csv'	File name used by the CSV export.
InlineEdit	Boolean	False	Lets the user edit a cell by double-clicking it.
EditMode	TsgcHTML-GridEditMode	geCell	Inline edit granularity: geCell or geRow.
ShowGroup-ing	Boolean	False	Groups rows by GroupByColumn with a group header row.
GroupByCol-umn	string	''	Name of the column used to group rows.
ColumnRe-order	Boolean	False	Allows the user to drag columns to reorder them.
VirtualScroll	Boolean	False	Renders only VisibleRows and loads more on scroll.
VirtualScrol-lURL	string	''	URL the virtual scroller requests for more rows (defaults to /grid/more).
VisibleRows	Integer	20	Number of rows rendered before virtual scrolling loads more.

XLSX Export

What it does: Set *ExportXLSX* to True and the grid renders an extra button, captioned by *ExportXLSXText*, that simply navigates the browser to *ExportURL*. The workbook itself is never assembled by client JavaScript: your application answers that URL with the bytes written by **SaveToXLSXStream**, which puts the current columns and rows into a worksheet named by *ExportSheetName*. Cells that hold a number are written as numbers (the detection accepts both '.' and ',' as the decimal separator, so it does not depend on the locale of the server), and everything else is written as text.

Property	Type	Default	Description
ExportXLSX	Boolean	False	Renders an Export XLSX button that navigates to ExportURL.
ExportURL	string	''	URL the button navigates to, where the host serves the workbook (defaults to /grid/export.xlsx).
ExportXLSX-Text	string	'Export XLSX'	Caption of the XLSX button.

ExportSheetName	string	'Sheet1'	Worksheet name inside the generated workbook.

```
// Render the button
oGrid->ExportXLSX = true;
oGrid->ExportURL = "/customers/export.xlsx";
oGrid->ExportSheetName = "Customers";

// Serve the workbook from that URL
AResponseInfo->ContentStream = new TMemoryStream();
oGrid->SaveToXLSXStream(AResponseInfo->ContentStream);
AResponseInfo->ContentType =
  "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet";
```

Tree Data

What it does: Set *TreeMode* to True, and name the column that holds the id of each row in *TreeIdField* and the column that holds the id of its parent in *TreeParentIdField*, and the grid renders the rows as a hierarchy instead of a flat list: children are indented under their parent, and every parent gets a chevron that expands and collapses its branch. A row with an empty parent id, or with a parent id that matches no other row, is a root, so orphan rows are never lost. *TreeExpandAll* decides whether the tree starts open. If either field name does not match a column, the grid falls back to a plain flat table. **LoadFromDataSetTree** loads a dataset and sets the three properties in one call.

Property	Type	Default	Description
TreeMode	Boolean	False	Renders the rows as an indented, expandable hierarchy.
TreeIdField	string	''	Name of the column holding the id of each row.
TreeParentIdField	string	''	Name of the column holding the id of the parent row.
TreeExpandAll	Boolean	True	Starts the tree fully expanded.

```
// qryEmployees has the fields EmployeeId, ManagerId, Name, Department
oGrid->LoadFromDataSetTree(qryEmployees, "EmployeeId", "ManagerId");
oGrid->TreeExpandAll = false;
html = oGrid->HTML;
```

Master-Detail Rows

What it does: Set *MasterDetail* to True and the grid prepends an expander column, and emits behind every record a hidden row that spans the whole table. Clicking the chevron opens and closes that row. Its content comes from the **OnGetDetailHTML** event, raised once per rendered row with the index of the row. The string you return is emitted **raw**, so the host application must HTML-encode any user data it injects there. Tree rendering takes precedence: when the grid renders as a tree, the expander column is not added.

Property	Type	Default	Description
MasterDetail	Boolean	False	Adds an expander column that reveals a full-width detail row filled by OnGetDetailHTML.

Saved Views and Column Visibility

What it does: Set *ShowColumnsMenu* to True and the grid renders a dropdown, captioned by *ColumnsText*, with one check box per column, so the user can hide the columns that do not interest him. Set *SavedViews* to True and the sort column and direction, the text typed in the filter box and the list of hidden columns are written to the browser localStorage under *PersistKey*, and restored the next time the grid loads. When *PersistKey* is empty the key is the table id followed by -view. Nothing is stored on the server, so each user keeps his own view.

Property	Type	Default	Description
SavedViews	Boolean	False	Persists sort, filter text and column visibility to localStorage and restores them on load.
PersistKey	string	''	localStorage key used by SavedViews (defaults to the table id plus -view).
ShowColumnsMenu	Boolean	False	Renders a Columns dropdown to toggle column visibility.
ColumnsText	string	'Columns'	Caption of that dropdown.

AI Query Bar

What it does: When *AIQueryEnabled* is True the grid renders a query form above the table. *AIQueryPlaceholder*, *AIQueryButtonText* and *AIQueryButtonStyle* style the bar. The submitted text reaches the server, where you call **ProcessAIQuery**.

With an *AIAssistant* assigned, ProcessAIQuery turns the question into a validated operation set and the grid applies it to its own rows: filters, sorts, a grouping, a limit and contextual highlights. **OnAIQuery** still wins, so a host that answers the event itself behaves exactly as before. The question and the operation set are kept in **QueryState**, and the bar also renders the interpreted as chip and a clear control. See AI Data Pack.

Property	Type	Default	Description
AIQueryEnabled	Boolean	False	Renders a natural-language query bar above the grid.
AIQueryPlaceholder	string	'Ask about your data...'	Placeholder text of the query input.

AIQueryButton-Text	string	'Ask AI'	Caption of the submit button.
AIQueryButton-Style	TsgcHTML-ButtonStyle	bsOutlinePri-mary	Bootstrap button style of the submit button.
AIAssistant	TsgcHTMLA-IAssistant	nil	The assistant that answers the bar and the col-umn assistant. Without it no AI markup is ren-dered.
AIPersistState	Boolean	True	Keeps the query state in the request session when there is one, instead of in a hidden input of the bar.

AI Column Assistant

What it does: a column whose *AIMode* is not acOff renders a suggest control in its header. **ProcessAIColumnVal-ues** asks the assistant for a value for every visible row in one call, and the answers become **AIProposals**, each shown next to the current cell value with an Accept and a Reject control. Only what a human accepts is reported, through **OnAIColumnValues**, and the grid writes nothing into your data.

The call sends row data, so it refuses while the assistant has AllowRowDataToProvider False. Filling AIProposals yourself shows the same Accept and Reject half with no provider at all.

Property	Type	De-fault	Description
AIPrompt	string	''	Of a column. What the assistant is asked to produce for it, for instance classify the sentiment as positive, neutral or negative.
AIMode	TsgcHTM-LAIColumn-Mode	acOff	Of a column. acOff, acOnDemand (the header renders the sug-gest control) or acAuto (the same control, and the host runs the suggestion as soon as the rows are loaded).
AIPropos-als	TsgcHTM-LAIProposals	(emp-ty)	Public. The pending proposals of one column: a Column name, and a RowIndex and a Value per item.
QueryS-tate	TsgcHTMLAI-QueryState	(emp-ty)	Public. The question, the validated operation set and the inter-preted as text.

Paging

What it does: Set *PageSize* above 0 to page the rows. With *PagingURL* empty the browser pages the rendered rows after the filter and the sort (detail rows follow their master, group headers are kept). With *PagingURL* set the server pages: **Rows** holds only the current page, the host sets *TotalRows* and *CurrentPage*, and every pager but-ton asks *PagingURL* for another page, replacing the grid wrapper `<TableID>-pg` with the HTML the endpoint returns. In that mode sorting and filtering are done by the server too. *VirtualScroll* wins over paging, and client paging is not applied in TreeMode.

Property	Type	Default	Description
PageSize	Integer	0	Rows per page. 0 turns paging off.
PagingURL	string	''	Endpoint of server side paging. Empty means client side paging.
TotalRows	Integer	0	Size of the whole data set for server paging. 0 means Rows.Count.
Current-Page	Integer	1	Current page, 1 based.
ShowPager	Boolean	True	Shows the first, previous, page number, next and last buttons plus the "x-y of N" text.

```
// server side paging: the endpoint renders one page
oGrid->PageSize = 20;
oGrid->PagingURL = "/orders/page";
oGrid->CurrentPage = vPage;
oGrid->TotalRows = vTotal;
```

Frozen Columns, Resizing and Multi-Sort

What it does: A column with *Frozen* = True stays pinned to the left edge while the table scrolls horizontally; several frozen columns stack in their visual order. *ColumnResize* adds a drag handle to every header (minimum width 40px, widths stored with SavedViews). *MultiSort*, together with ShowSort, lets Shift+click add a column to the sort list.

Property	Type	Default	Description
Frozen	Boolean	False	Of a column. Pins the column to the left edge.
ColumnResize	Boolean	False	Lets the user resize columns by dragging the header edge.
MultiSort	Boolean	False	Shift+click adds a column to the sort instead of replacing it.

Row Selection and Bulk Actions

What it does: *SelectionMode* adds a checkbox column (gsmSingle or gsmMultiple, with a select-all checkbox). Each row is identified by *KeyField*, or by its index. With gsmMultiple and *BulkActions*, a bar shows "N selected" and one button per action while rows are selected; a click posts the action name and the selected keys to *BulkActionURL*. Without server paging the table then raises the DOM event `sgcGridBulkAction` (detail: action, keys, successful). With server paging the endpoint answers with the grid HTML and the selection survives page changes.

Property	Type	Default	Description
SelectionMode	TsgcHTMLGridSelectionMode	gsmNone	gsmNone, gsmSingle or gsmMultiple. Ignored in TreeMode.
KeyField	string	''	Column whose value identifies a row. Empty means the row index.
BulkActions	TsgcHTMLGridBulkActions	(empty)	Buttons of the bulk bar: Caption, ActionName, ButtonStyle (bsOutlineSecondary) and an optional Confirm text.
BulkActionURL	string	''	Endpoint that receives the bulk action POST.

Typed Columns, Editors and Validation

What it does: Values in **Rows** are always invariant (numbers with a '.' decimal separator, dates as yyyy-mm-dd, booleans as 1/0). A column *DataType* other than gdtText renders them with *DisplayFormat* and the FormatSettings of the process, and keeps the raw value in the `data-v` attribute, which the browser uses to sort, filter and export. With InlineEdit, *EditorType* picks a number, date, datetime, checkbox, select (*EditorItems*) or lookup (*LookupURL*) editor, and *Required*, *MinValue*, *MaxValue*, *Pattern* and *MaxLength* validate the value in the browser before it is posted. *ValidationMessage* replaces the locale messages.

Property	Type	Default	Description
DataType	TsgcHTMLGridDataType	gdtText	Of a column. gdtText, gdtNumber, gdtCurrency, gdtDate, gdtDateTime or gdtBoolean.
DisplayFormat	string	''	Of a column. FormatFloat or FormatDateTime picture.
EditorType	TsgcHTMLGridEditorType	getAuto	Of a column. getAuto, getText, getNumber, getDate, getSelect, getCheckbox or getLookup.
EditorItems	TStringList	(empty)	Of a column. Options of getSelect, value=Caption or a plain value per line.
LookupURL	string	''	Of a column. getLookup suggestions, requested with the parameter q.
Required	Boolean	False	Of a column. Rejects an empty value.
MinValue / MaxValue	string	''	Of a column. Invariant limits, a number or yyyy-mm-dd.
Pattern	string	''	Of a column. JavaScript regular expression the whole value must match.
MaxLength	Integer	0	Of a column. Maximum length, 0 is no limit.

Validation-Message	string	''	Of a column. Replaces the locale validation messages.

Summaries, Conditional Formatting and Renderers

What it does: A column *Summary* (gsSum, gsAvg, gsCount, gsMin, gsMax) adds a footer row with the aggregate, formatted with *SummaryFormat*, and recomputed in the browser after every filter and sort (not with server paging or in TreeMode). When the grid is grouped, every group also gets a subtotal row; grouping needs *ShowGrouping* = True, GroupByColumn alone draws no groups. *FooterValues* lets the host supply the footer text instead, for instance the totals of the whole data set with server paging; when it is not empty it replaces the whole computed footer row. *FormatRules* add CSS classes or styles to cells or rows whose raw value matches a *RuleOperator*. A column *Renderer* draws the value as a progress bar, a badge, a sparkline or a check mark, configured by *RendererOptions*.

Property	Type	Default	Description
Summary	TsgcHTMLGrid-Summary	gsNone	Of a column. Aggregate shown in the footer and in group subtotals.
Summary-Format	string	''	Of a column. Picture of the aggregate, defaults to DisplayFormat.
Renderer	TsgcHTMLGridRenderer	grNone	Of a column. grrNone, grrProgress, grrBadge, grrSparkline or grrBoolean.
RendererOptions	string	''	Of a column. key=value;... options, for instance max=100;style=success.
FormatRules	TsgcHTMLGrid-FormatRules	(empty)	Conditional formatting: ColumnName, RuleOperator, Value, Value2, CSSClass, Style, ApplyToRow.
FooterValues	TStringList	(empty)	ColumnName=text lines that replace the whole computed footer row.

```
TsgcHTMLGridColumn *col = oGrid->Columns->Add();
col->Name = "Amount";
col->DataType = gdtCurrency;
col->Summary = gsSum;
TsgcHTMLGridFormatRule *rule = oGrid->FormatRules->Add();
rule->ColumnName = "Amount";
rule->RuleOperator = groGreater;
rule->Value = "1000";
rule->CSSClass = "table-success";
```

Keyboard Navigation and Live Rows

What it does: *KeyboardNavigation* turns the table into an ARIA grid with one focusable cell: arrows, Home/End, Ctrl+Home/End and PageUp/PageDown move the focus, Enter or F2 edits (with InlineEdit), Tab and Enter save and move, Escape cancels, and Space toggles the row selection. *LiveRows* gives every row a stable id so **GetRowFragmentHTML**, **GetRowInsertFragmentHTML** and **GetRowDeleteFragmentHTML** can update, insert or delete a single row through an htmx out of band fragment, for instance pushed with the sgcHTMX server. The browser then applies again the sort, filter, page, summaries, selection and frozen columns. Do not mix an insert fragment with update or delete fragments in the same push. LiveRows is not applied in TreeMode.

Property	Type	Default	Description
KeyboardNavigation	Boolean	False	Keyboard navigation with a roving focused cell.
LiveRows	Boolean	False	Row ids for the single row fragment functions.

Server Request Parameters

Several features call back to the host, which must read these parameters and answer:

Parameter	Sent by	Content
page	Pager (GET PagingURL), bulk actions with server paging	Requested page, 1 based. A sort or filter change requests page 1.
pageSize	Pager, bulk actions with server paging	The PageSize.
sort	Pager, bulk actions with server paging	Sort list as `Name:asc,Other:desc` (column Names), empty when unsorted.
filter	Pager, bulk actions with server paging	Text of the filter box. Typing requests page 1 after a 300 ms pause.
selected	Pager, when SelectionMode is not gsmNone	Selected keys, comma separated, each key URL encoded.
action	Bulk action (POST BulkActionURL)	The ActionName of the clicked button.
keys	Bulk action	Selected keys, comma separated, each key URL encoded.
q	getLookup editor (GET LookupURL)	Text typed so far. Answer with `<option value="...">` elements.
gridEdit message: table, row, col, value	Inline editor, through the sgcHTMX WebSocket	`row` is the index in Rows, `col` the index in Columns and `value` the raw invariant value.

The paging and bulk endpoints answer with the grid **HTML** rendered for the requested page (Rows, TotalRows, CurrentPage and, when needed, FooterValues set). Always validate these values on the server.

Localization

The fixed texts of these features come from the shared locale (see Localization): **GridPageOfText** (the “of” of the pager), **GridSelectedText** (“%d selected” of the bulk bar), **GridRequiredText**, **GridMinValueText**, **GridMaxValueText** and **GridInvalidFormatText** (the inline editor validation messages), plus the pager labels FirstPageLabel, PrevPageLabel, NextPageLabel and LastPageLabel and CancelText of the bulk confirm step.

Demo

The **06.Grid** demo (Demos\60.HTML\01.RunTime\06.Grid) shows these features in five tabs: *Paging & Frozen*, *Selection*, *Typed & Edit*, *Summaries* and *Keyboard & Live*.

Output

The inherited **HTML** property returns the optional AI query bar, the table (wrapped in a responsive or virtual-scroll container) and, when any interactive feature is enabled, an inline initialization `<script>`. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 57 properties available on the component, plus the column properties.
- Methods — 17 public methods.
- Events — 3 published events.

TsgcHTMLComponent_DataTable

TsgcHTMLComponent_DataTable wraps a grid and a pagination bar inside a Bootstrap card, adding a toolbar with a title, search box, page-size selector and export button plus a footer that shows the row count and page links.

Introduction

TsgcHTMLComponent_DataTable is a higher-level data table built on top of **TsgcHTMLComponent_Grid** and **TsgcHTMLComponent_Pagination**. It owns one of each and exposes them read-only through the **Grid** and **Pagination** properties, so you configure the table columns and styling on **Grid** and the page navigation on **Pagination**.

The component renders a Bootstrap `card` containing a toolbar (title, optional search form, page-size selector and export button), the grid, and a footer that shows the visible row count and the pagination control. Read the inherited **HTML** property to get the complete markup.

Quick Start

1. Drop a **TsgcHTMLComponent_DataTable** component on the form (or create it at runtime).

2. Set the **Title**, then call **LoadFromDataSet** with your dataset and a page size.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_DataTable *oTable = new TsgcHTMLComponent_DataTable(NULL);
try
{
  oTable->Title = "Customers";
  oTable->ShowSearch = true;
  oTable->LoadFromDataSet(qryCustomers, 25);
  html = oTable->HTML;
}
__finally
{
  delete oTable;
}
```

Grid and Pagination

What it does: The read-only *Grid* property is the underlying **TsgcHTMLComponent_Grid**; use it to set the columns, styling and interactive options. The read-only *Pagination* property is the underlying **TsgcHTMLComponent_Pagination**; **LoadFromDataSet** sets its total items, page size and current page for you, and you can adjust its appearance afterwards.

Property	Type	Default	Description
Grid	TsgcHTMLComponent_Grid		The underlying grid that renders the table (read-only).

Pagina-tion	TsgcHTMLComponent_Pagination		The underlying pagination control shown in the footer (read-only).
TableID	string	'sgc-DataT-able'	The id of the card; the grid is given this id plus the suffix _grid.
Title	string	''	Optional heading shown in the toolbar.

```
oTable->Grid->Striped = true;
oTable->Pagination->MaxVisible = 7;
```

Toolbar

What it does: The toolbar flags *ShowSearch*, *ShowPageSize* and *ShowExport* turn the search box, page-size selector and export button on or off. *SearchPlaceholder* and *SearchAction* configure the search form, *PageSizes* lists the page-size options, and *ToolbarClass* overrides the toolbar row class.

Property	Type	Default	Description
ShowSearch	Boolean	True	Shows a search box in the toolbar.
SearchPlace-holder	string	'Search...'	Placeholder text of the search input.
SearchAction	string	''	When set, the search becomes a GET form posting to this URL.
ShowPageSize	Boolean	True	Shows a page-size selector in the toolbar.
PageSizes	string	'10,25,50,100'	Comma-separated list of page-size options.
ShowExport	Boolean	False	Shows an export button in the toolbar.
ToolbarClass	string	''	Overrides the CSS class of the toolbar row.

XLSX Export and Column Visibility

What it does: *ShowExportXLSX* adds an Export XLSX button to the toolbar, placed after the export button and before the search box. The button is a link pointing at *ExportXLSXURL* (`/grid/export.xlsx` when empty) and captioned with *ExportXLSXText*; the workbook is not built in the browser, so the host application must answer that URL with the bytes of **Grid.SaveToXLSXStream**. *ShowColumnsMenu* surfaces the inner grid's Columns dropdown, which is drawn above the grid table and holds one checkbox per column so the user can hide and show columns; the value is forwarded to **Grid.ShowColumnsMenu** when the table renders.

Property	Type	Default	Description
ShowExportXLSX	Boolean	False	Adds an Export XLSX button to the toolbar.
ExportXLSXURL	string	''	URL the Export XLSX button points at.
ExportXLSXText	string	'Export XLSX'	Caption of the Export XLSX button.
ShowColumnsMenu	Boolean	False	Shows the Columns dropdown that toggles column visibility.

```
oTable->ShowExportXLSX = true;
oTable->ExportXLSXURL = "/customers/export.xlsx";
oTable->ExportXLSXText = "Export XLSX";
oTable->ShowColumnsMenu = true;

// the route that answers /customers/export.xlsx writes the workbook:
oTable->Grid->SaveToXLSXStream(aResponseStream);
```

Footer

What it does: When *ShowRowCount* is True the footer shows how many rows are displayed alongside the pagination control returned by **Pagination**.

Property	Type	Default	Description
ShowRowCount	Boolean	True	Shows the visible row count in the footer.

AI Query Bar

What it does: *AIQueryEnabled* renders a natural language query bar above the table, answered by the *AIAssistant* you assign. Both are pushed into the grid the table wraps, so one assistant serves both components and the host routes one action for both. See AI Data Pack.

The bar belongs to that inner grid, so the `table` field of its actions carries `<SafeID(TableID)>_grid`. Answer the action with **LoadAIQueryState**, **ProcessAIQuery** and, for a bound table, **LoadFromDataSource**.

Property	Type	Default	Description
AIQueryEnabled	Boolean	False	Renders the query bar. Off by default, and then nothing at all is rendered.
AIQueryPlaceholder	string	'Ask about your data...'	Placeholder of the query input.

AIAssistant	TsgcHTMLA-IAssistant	nil	The assistant that answers the bar.
QueryState	TsgcHTM-LAIQueryS-tate	(empty)	Public. The query state of the grid the table wraps.

Output

The inherited **HTML** property returns a Bootstrap `card` wrapping the toolbar, the grid markup and the footer. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 20 properties available on the component.
- Methods — 5 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Calendar

TsgcHTMLComponent_Calendar renders a month calendar as a Bootstrap table, with optional navigation arrows, a highlighted current day and a colored dot on days that carry an event.

Introduction

TsgcHTMLComponent_Calendar draws a single month grid inside a Bootstrap card. You set the **Year** and **Month** (both default to the current date), add day events to the **Events** collection, then read the inherited **HTML** property to get the complete markup.

Each cell shows the day number; today is highlighted when **HighlightToday** is on, and any day that has a matching event in the **Events** collection gets a small colored dot. Previous and next links are emitted when **ShowNavigation** is on and you provide the **PrevURL** and **NextURL**.

Quick Start

1. Drop a **TsgcHTMLComponent_Calendar** component on the form (or create it at runtime).

2. Set **Year** and **Month**, then add one or more day events to the **Events** collection (or call **LoadFromDataSet**).

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Calendar *oCalendar = new TsgcHTMLComponent_Calendar(NULL);
try
{
  oCalendar->Year = 2026;
  oCalendar->Month = 6;
  TsgcHTMLCalendarEvent *evt = oCalendar->Events->Add();
  evt->Day = 18;
  evt->Title = "Release";
  evt->ColorStyle = hcSuccess;
  html = oCalendar->HTML;
}
__finally
{
  delete oCalendar;
}
```

Month and Events

What it does: *Year* and *Month* select the month drawn. *Events* is a collection of **TsgcHTMLCalendarEvent** items, each with a *Day*, *Title*, a free-form *Color* and a *ColorStyle* from the Bootstrap palette. A day that has a matching event shows a colored dot below the day number. Note that *Events* is a data collection, not a Delphi event; populate it directly or with **LoadFromDataSet**.

Property	Type	Default	Description
Year	Integer	current year	The year drawn by the calendar.

Month	Integer	current month	The month drawn (1–12).
Events	TsgcHTMLCalendarEvents		Collection of TsgcHTMLCalendarEvent items (a data collection, not a Delphi event).
EventDotSize	Integer	6	Diameter in pixels of the event dot.

```
// Build events from a query (date field + title field)
oCalendar->LoadFromDataSet(qryMeetings, "MeetingDate", "Subject");
```

Navigation and Appearance

What it does: *ShowNavigation* draws previous and next links above the grid, pointing at *PrevURL* and *NextURL*. *HighlightToday* marks the current day with the *TodayClass*. *NavButtonClass* styles the navigation links, *CalendarID* sets the card id and *TableClass* overrides the calendar table class.

Property	Type	Default	Description
ShowNavigation	Boolean	True	Shows previous/next navigation links above the calendar.
PrevURL	string	''	URL of the previous-month link.
NextURL	string	''	URL of the next-month link.
NavButtonClass	string	'btn btn-sm btn-outline-secondary'	CSS class of the navigation links.
HighlightToday	Boolean	True	Highlights the current day.
TodayClass	string	'bg-primary text-white rounded'	CSS class applied to today's cell.
CalendarID	string	'sgcCalendar'	The id of the calendar card.
TableClass	string	'table table-sm text-center mb-0'	CSS class of the calendar table.

Output

The inherited **HTML** property returns a Bootstrap `card` with an optional navigation header and a `<table>` whose body lays out the month one week per row. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 12 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Scheduler

TsgcHTMLComponent_Scheduler renders a month or week scheduler as a Bootstrap table, placing colored event blocks on the days they span.

Introduction

TsgcHTMLComponent_Scheduler draws a calendar-style scheduler inside a Bootstrap card. You pick the **View** (month or week) and the **CurrentDate** that anchors it, add appointments to the **Events** collection, then read the inherited **HTML** property to get the complete markup, including the scheduler's own CSS.

Each event is drawn as a colored block on every day it covers. The month view shows the whole month one week per row; the week view shows an hourly grid from **StartHour** to **EndHour**. The day view shares the week-view layout.

Quick Start

1. Drop a **TsgcHTMLComponent_Scheduler** component on the form (or create it at runtime).

2. Set the **View** and **CurrentDate**, then add events with **AddEvent** (or call **LoadFromDataSet**).

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Scheduler *oScheduler = new TsgcHTMLComponent_Scheduler(NULL);
try
{
  oScheduler->View = svWeek;
  oScheduler->CurrentDate = Now();
  oScheduler->AddEvent("Stand-up", EncodeDate(2026, 6, 18) + EncodeTime(9, 0, 0, 0),
    EncodeDate(2026, 6, 18) + EncodeTime(9, 30, 0, 0), hcPrimary);
  html = oScheduler->HTML;
}
__finally
{
  delete oScheduler;
}
```

View

What it does: *View* selects month or week (the day view uses the week layout). *CurrentDate* is the date the view is built around: the month it belongs to in month view, or the week it falls in for week view. *StartHour* and *EndHour* bound the hourly rows of the week view, and *SchedulerID* sets the card id.

Property	Type	Default	Description
View	TsgcHTMLSchedulerView	svMonth	The scheduler layout: svDay, svWeek or svMonth.

CurrentDate	TDateTime	now	The date the view is anchored to.
StartHour	Integer	8	First hour shown in the week view.
EndHour	Integer	18	Last hour shown in the week view.
SchedulerID	string	'sgcScheduler'	The id of the scheduler card.

Events

What it does: *Events* is a collection of **TsgcHTMLSchedulerEvent** items, each with a *Title*, *StartDate*, *EndDate*, a Bootstrap *Color*, an optional *Description* (shown as a tooltip) and an *AllDay* flag. Add an appointment with **AddEvent**, or fill the collection from a dataset with **LoadFromDataSet**. Note that *Events* is a data collection, not a Delphi event.

Property	Type	Default	Description
Events	TsgcHTMLSchedulerEvents		Collection of TsgcHTMLSchedulerEvent items (a data collection, not a Delphi event).

```
// Build events from a query (title + start + end fields)
oScheduler->LoadFromDataSet(qryBookings, "Subject", "StartTime", "EndTime");
```

Output

The inherited **HTML** property returns a Bootstrap `card` with a header showing the current month and year and a `<table>` laying out the chosen view. The component contributes its own CSS through the inherited **CSS** mechanism, so the event blocks and today highlight render correctly. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 6 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Timeline

TsgcHTMLComponent_Timeline renders a vertical timeline of dated events, each with a title, body text, a colored dot and an optional timestamp, from items you add directly or load from a dataset.

Introduction

TsgcHTMLComponent_Timeline emits a Bootstrap-styled vertical timeline: a connecting line down the left edge with one card per event. Each event is a **TsgcHTMLTimelineItem** exposing a *Title*, *Content*, *Timestamp*, a per-item *Color* (or *ColorStyle* theme color), and an optional *Icon*. The component builds the markup for you and returns it through its inherited **HTML** property, and emits the matching CSS through **CSS**.

The line color, line width, dot size and timestamp color are all set once on the component and apply to the whole timeline, so you control the overall look without styling each item.

Quick Start

1. Drop a **TsgcHTMLComponent_Timeline** component on the form (or create it at runtime).

2. Add one **TsgcHTMLTimelineItem** per event through the **Items** collection (or call **LoadFromDataSet**).

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Timeline *oTimeline = new TsgcHTMLComponent_Timeline(NULL);
try
{
  TsgcHTMLTimelineItem *oItem = oTimeline->Items->Add();
  oItem->Timestamp = "09:00";
  oItem->Title = "Order received";
  oItem->Content = "Customer placed order #1042.";
  oItem->ColorStyle = hcSuccess;
  oItem = oTimeline->Items->Add();
  oItem->Timestamp = "11:30";
  oItem->Title = "Shipped";
  oItem->Content = "Handed to the carrier.";
  oItem->ColorStyle = hcPrimary;
  html = oTimeline->HTML;
}
__finally
{
  delete oTimeline;
}
```

Events

What it does: Each entry on the timeline is a **TsgcHTMLTimelineItem** held in the *Items* collection. An item carries a *Title*, a *Content* body, a *Timestamp* label, an optional *Icon* name, and a color. Leave *Color* empty to use the theme color from *ColorStyle*, or set a CSS color string to override it.

Property	Type	Default	Description

Items	TsgcHTMLTime-lineItems	–	Collection of TsgcHTMLTimelineItem entries, one per event.
TimelineID	string	'sgcTime-line'	The id of the generated container .

```
TsgcHTMLTimelineItem *oItem = oTimeline->Items->Add();
oItem->Timestamp = "2026-01-15";
oItem->Title = "Milestone reached";
oItem->Content = "Beta released to early adopters.";
oItem->ColorStyle = hcInfo;
```

Appearance

What it does: *LineColor* and *LineColorStyle* set the color of the connecting line, *LineWidth* its thickness and *DotSize* the diameter of each event dot. *TimeColor* and *TimeColorStyle* color the timestamp text. The *Color* properties take a CSS color string and win when set; otherwise the matching *ColorStyle* theme color is used.

Property	Type	Default	Description
LineColor	string	''	CSS color of the connecting line. Empty falls back to Line-ColorStyle.
LineCol-orStyle	TsgcHTML-Color	hcLight	Theme color of the line when LineColor is empty.
LineWidth	Integer	2	Width of the connecting line, in pixels.
DotSize	Integer	18	Diameter of each event dot, in pixels.
TimeColor	string	''	CSS color of the timestamp text. Empty falls back to Time-ColorStyle.
TimeCol-orStyle	TsgcHTML-Color	hcSec-ondary	Theme color of the timestamp when TimeColor is empty.

```
oTimeline->LineColorStyle = hcPrimary;
oTimeline->LineWidth = 3;
oTimeline->DotSize = 22;
oTimeline->TimeColorStyle = hcMuted;
```

Loading from a Dataset

What it does: LoadFromDataSet clears the current items and adds one entry per row, mapping a title field and optional content and timestamp fields. It is the quickest way to drive the timeline from a query.

```
oTimeline->LoadFromDataSet(qryHistory, "Action", "Notes", "EventDate");
```

Output

The inherited **HTML** property returns the timeline `<div>` with one card per item, and **CSS** returns the styles that position the line and dots. Serve these directly, or assign the HTML to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 8 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_KanbanBoard

TsgcHTMLComponent_KanbanBoard renders a Kanban board: a row of columns, each holding a stack of cards, with optional drag-and-drop between columns. You add columns and cards directly or load them from a dataset.

Introduction

TsgcHTMLComponent_KanbanBoard emits a horizontal set of columns, each with a colored header, a card count badge and a body that holds cards. A column is a **TsgcHTMLKanbanColumn** with its own *Title*, *Color* and *Cards* collection; a card is a **TsgcHTMLKanbanCard** with a *Title*, *Description*, *Assignee*, a *Tag* with its *TagColor*, and a per-card *Color*. The component builds the markup for you and returns it through its inherited **HTML** property, and emits the matching CSS through **CSS**.

When **DragEnabled** is True the board also emits a SortableJS script so cards can be dragged between columns in the browser.

Quick Start

1. Drop a **TsgcHTMLComponent_KanbanBoard** component on the form (or create it at runtime).

2. Add columns with **AddColumn** and add cards to each column with the column's **AddCard** method (or call **LoadFromDataSet**).

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_KanbanBoard *oBoard = new TsgcHTMLComponent_KanbanBoard(NULL);
try
{
  TsgcHTMLKanbanColumn *oTodo = oBoard->AddColumn("To Do", hcSecondary);
  oTodo->AddCard("Draft spec", "Outline the API.", hcLight, "Alex");
  oTodo->AddCard("Set up CI", "", hcLight, "Sam");
  TsgcHTMLKanbanColumn *oDoing = oBoard->AddColumn("In Progress", hcPrimary);
  oDoing->AddCard("Build parser", "Tokenizer first.", hcLight, "Alex");
  html = oBoard->HTML;
}
__finally
{
  delete oBoard;
}
```

Columns and Cards

What it does: The board holds a *Columns* collection of **TsgcHTMLKanbanColumn**. Each column has a *Title*, a header *Color*, a *ColumnID* and a *Cards* collection of **TsgcHTMLKanbanCard**. A card has a *Title*, *Description*, *Assignee*, a *Tag* with its *TagColor*, a card *Color* and a *CardID*. Add columns with **AddColumn** and cards with the column's **AddCard** method. The URL of a card is sanitized before it reaches the click handler of the card, so an unsafe scheme such as `javascript:` becomes `#`.

Property	Type	Default	Description

Columns	TsgcHTMLKanban-Columns	–	Collection of TsgcHTMLKanbanColumn items drawn left to right.
BoardID	string	'sgcKan-ban'	The id of the generated board .

Layout and Drag

What it does: *Height* caps the board height and makes it scroll vertically. *DragEnabled* emits the SortableJS script that lets users drag cards between columns in the browser. *DragUpdateURL* is sanitized and *DragUpdateExtra* is escaped for the JavaScript string they are written into.

Property	Type	Default	Description
Height	string	'500px'	Maximum board height; the board scrolls vertically beyond it. Empty removes the cap.
DragEn-abled	Boolean	False	When True emits the SortableJS script for drag-and-drop be-tween columns.
SortableJS-Path	string	(CDN URL)	URL of the SortableJS script emitted when DragEnabled is True. Point it at a local copy to avoid the CDN.

```
oBoard->Height = "600px";
oBoard->DragEnabled = true;
```

Loading from a Dataset

What it does: LoadFromDataSet clears the board and groups rows into columns by a group field, adding one card per row with a title and optional description and assignee. Columns are created on demand as new group values appear.

```
oBoard->LoadFromDataSet(qryTasks, "Status", "Task", "Notes", "Owner");
```

Card Details

What it does: besides its title, description, assignee and tags, a card can show extra badges (*Tags*), a due date that turns red when it is overdue (*DueDate*), a priority dot (*Priority*), a thin progress bar (*Progress*), an avatar (*AvatarURL*) and a checklist counter (*ChecklistDone*, *ChecklistTotal*). With *ShowAssigneeInitials*, a card without avatar shows the initials of the assignee. See Columns for every card and column member.

```
TsgcHTMLKanbanColumn *oCol = oBoard->AddColumn("To do");
TsgcHTMLKanbanCard *oCard = oCol->Cards->Add();
oCard->Title = "Fix login";
oCard->CardID = "task-7";
oCard->DueDate = "2026-10-15";
oCard->Priority = kpHigh;
oCard->Progress = 40;
oCard->Tags->Add()->Text = "backend";
```

Quick Add and Edit

What it does: *QuickAdd* adds a small form at the end of every column that posts the title of a new card to *QuickAddURL*. *EditURL* opens a modal with the edit form of a card, loaded from the server, when the user clicks a card without *URL*. Both work through htmx.

Property	Type	Default	Description
QuickAdd	Boolean	False	Adds a small form at the end of every column to create a card by typing its title.
QuickAddURL	string	''	The URL the quick add form of every column posts to.
QuickAddText	string	'Add card'	The caption of the quick add button.
EditURL	string	''	The URL that loads the edit form of a card into a modal.
ShowAssigneeInitials	Boolean	False	Shows a circle with the initials of the assignee on cards without an avatar.

Swimlanes, WIP Limits and Collapsible Columns

What it does: *Swimlanes* splits the board in horizontal lanes (a card picks its lane with *SwimlaneID*). Every column can set a work in progress limit (*WIPLimit*), the columns its cards may be dropped into (*AllowedTargets*) and, with *CollapsibleColumns*, its initial *Collapsed* state.

Validate moves on the server. *AllowedTargets* only restricts the drop in the browser. The handler of *DragUpdateURL* receives `id`, `status` (the target column) and, with swimlanes, `swimlane`, and must check that the move is allowed before it saves it.

Property	Type	Default	Description
Swimlanes	TsgcHTMLKanbanSwimlanes	Empty collection	Horizontal lanes that split every column, for example by priority or by team.
SwimlaneOtherText	string	'Other'	The title of the lane that holds the cards with no known swimlane.

Collapsi-bleColumns	Boolean	False	Lets the user collapse a column to a narrow vertical strip.
ShowCard-Count	Boolean	False	Keeps the card count of every column up to date in the browser.

Search and Live Sync

What it does: *ShowSearch* renders a search box that filters the cards in the browser. *LiveSync* prepares the board for updates pushed from the server: the methods **GetCardFragmentHTML**, **GetCardInsertFragmentHTML**, **GetCardMoveFragmentHTML** and **GetCardRemoveFragmentHTML** build out-of-band fragments that **TsgcHTMX_Engine_Server** sends to one page (**PushFragment**) or to every page (**BroadcastFragment**), so every user sees a move as soon as it is saved.

Property	Type	Default	Description
ShowSearch	Boolean	False	Renders a search box that filters the cards in the browser.
SearchPlace-holder	string	'Search cards'	The placeholder of the search box.
LiveSync	Boolean	False	Prepares the board for live updates pushed from the server to every open page.

```
// the server has validated and saved the move
TsgcHTMLComponent_KanbanBoard *oBoard = BuildBoard();
try
{
  HTMXServer->BroadcastFragment(
    oBoard->GetCardMoveFragmentHTML(aCardID, aNewColumnID));
}
__finally
{
  delete oBoard;
}
```

Demo

The ticket board of the Helpdesk demo, `Demos\60.HTML\01.RunTime\09.Helpdesk`, uses swimlanes by priority, WIP limits, allowed targets validated again on the server, quick add, the edit modal, search and live sync between the open pages.

Output

The inherited **HTML** property returns the board `<div>` with one column per item and one card per entry, and **CSS** returns the styles that lay out the columns and cards. Serve these directly, or assign the HTML to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 17 properties available on the component.
- Methods — 6 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Gantt

TsgcHTMLComponent_Gantt renders an interactive Gantt chart: one bar per task on a shared date axis, with a zoomable timeline, arrows between dependent tasks, and bars the user can drag or resize in the browser, sending the new dates straight back to the server.

Introduction

TsgcHTMLComponent_Gantt emits a Bootstrap-styled Gantt chart. Each task is a **TsgcHTMLGanttTask** with a *Title*, a *StartDate* and *EndDate*, a *Progress* percentage, a theme *Color*, an optional *Assignee*, a stable *Id*, and a *DependsOn* list naming the tasks it waits for. The component finds the earliest start and latest end across all tasks, then places each bar by start offset and width as a percentage of that range. The result is returned through its inherited **HTML** property, with the layout CSS emitted through **CSS**.

The chart is not a static picture. The timeline can be zoomed by day, week or month, arrows connect each task to the tasks it depends on, and a marker shows today. The whole collection can be filled from a database in one call with **LoadFromDataSet**, and a chart bound through **DataSource** with **DataAutoRefresh** re-renders itself whenever the dataset changes.

Turn on **Draggable** or **Resizable** and the bars become editable. When the user drops a bar, the new dates travel back over the WebSocket connection with the action name `ganttTaskMoved`, the server hands them to **ProcessTaskMoved**, the task is updated, and **OnTaskMoved** fires so the change can be written to the database. That closes the loop between the browser and your Delphi code.

A footer shows the first and last dates of the range. When the chart has no tasks a placeholder card is emitted instead.

Quick Start

1. Drop a **TsgcHTMLComponent_Gantt** component on the form (or create it at runtime).

2. Add one task per row with **AddTask**, or with **AddTaskEx** when the task needs an id and dependencies, or fill the whole **Tasks** collection at once with **LoadFromDataSet**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Gantt *oGantt = new TsgcHTMLComponent_Gantt(NULL);
try
{
  oGantt->Title = "Release 2026.1";
  oGantt->AddTask("Design", EncodeDate(2026, 1, 5),  EncodeDate(2026, 1, 12), 100, hcSuccess, "Alex");
  oGantt->AddTask("Build",  EncodeDate(2026, 1, 13), EncodeDate(2026, 1, 26), 60,  hcPrimary, "Sam");
  oGantt->AddTask("Test",   EncodeDate(2026, 1, 27), EncodeDate(2026, 2, 3),  0,   hcWarning, "Lee");
  html = oGantt->HTML;
}
__finally
{
  delete oGantt;
}
```

Tasks

What it does: The chart holds a *Tasks* collection of **TsgcHTMLGanttTask**. Each task carries a *Title*, a *StartDate* and *EndDate*, a *Progress* value from 0 to 100, a theme *Color*, an optional *Assignee*, a stable *Id* and a *DependsOn* list. Add tasks with **AddTask**, with **AddTaskEx** when you need an id and dependencies, or by adding items to the collection directly.

Property	Type	Default	Description
Tasks	TsgcHTMLGantt-Tasks	–	Collection of TsgcHTMLGanttTask items, one bar per task.

```
oGantt->AddTask("Docs", EncodeDate(2026, 2, 4), EncodeDate(2026, 2, 9), 25, hcInfo, "Jo");
```

Appearance

What it does: *Title* sets the heading drawn above the bars; leave it empty to omit the heading. *GanttID* sets the id of the container card so the chart can be targeted by scripts and CSS.

Property	Type	Default	Description
Title	string	'Project Timeline'	Heading drawn above the bars. Empty omits the heading.
GanttID	string	'sgcGantt'	The id of the generated container .

Timeline and Appearance

What it does: *Zoom* chooses the scale of the timeline: one tick per day, one every seven days, or one per month. The same tick rule drives the vertical grid lines drawn by *ShowGrid*, so the header labels and the grid always line up. *ShowToday* draws a marker at the current date, but only when today falls inside the chart range. *ShowProgress* fills each bar to its *Progress* percentage, and skips tasks whose progress is still zero. *RowHeight* is the vertical pitch of the chart: it sets the height of each row, the bar inside it is inset by three pixels top and bottom.

Property	Type	Default	Description
Zoom	TsgcHTMLGanttZoom	gzWeek	Timeline scale: gzDay, gzWeek or gzMonth.
RowHeight	Integer	32	Pixel height of a task row.
ShowToday	Boolean	True	Draws a today marker line on the timeline.
ShowGrid	Boolean	True	Draws the vertical grid lines of the timeline scale.
ShowProgress	Boolean	True	Fills each bar to its Progress percentage.

```
oGantt->Zoom = gzMonth;
oGantt->RowHeight = 24;
oGantt->ShowToday = true;
oGantt->ShowGrid = true;
oGantt->ShowProgress = true;
```

Dependencies

What it does: Give each task a stable *Id*, then list the ids a task waits for in its *DependsOn*, separated by commas. With *ShowDependencies* True the component overlays an SVG layer and draws one arrow per dependency, from the right edge of each predecessor bar to the left edge of the dependent bar. Ids are matched trimmed and without case sensitivity; ids that match no task, and a task that depends on itself, are skipped silently. If no dependency resolves, no SVG layer is emitted. **AddTaskEx** is the shortcut that sets the id and the dependency list in one call and returns the created task.

Property	Type	Default	Description
ShowDependencies	Boolean	True	Draws the connector lines between a task and the tasks listed in its DependsOn.
Tasks[i].Id	string	–	Stable identifier of a task, referenced by DependsOn and by the drag payload.
Tasks[i].DependsOn	string	–	Comma-separated list of the task ids this task depends on.

```
oGantt->ShowDependencies = true;
oGantt->AddTaskEx("design", "Design", EncodeDate(2026, 1, 5),  EncodeDate(2026, 1, 12), "", 100, hcSuccess, '
oGantt->AddTaskEx("build",  "Build",  EncodeDate(2026, 1, 13), EncodeDate(2026, 1, 26), "design", 60, hcPrima
oGantt->AddTaskEx("test",   "Test",   EncodeDate(2026, 1, 27), EncodeDate(2026, 2, 3),  "design,build", 0, hc
```

Drag and Resize

What it does: With *Draggable* True the user can drag a whole bar along the timeline, keeping its duration and shifting both dates by the same number of days. With *Resizable* True each bar gets a handle on its left edge and one on its right edge, so the start or the end date can be changed on its own. Both are clamped to the chart range, and the resize handles cannot cross each other. The two properties are independent.

Either of them makes the component emit a small drag script and a hidden form marked `data-sgc-ws-send`. When the user drops the bar, and only if the dates really changed, the form is submitted, and the sgcHTMX client sends its fields over the WebSocket connection: the action name `ganttTaskMoved`, the gantt id, the task id, and the new `start` and `end` dates as ISO `yyyy-mm-dd` strings. A DOM CustomEvent named `sgcGantt:taskMoved` is dispatched as well, so the page can react locally without waiting for the server.

On the server, the host receives the message in **OnHTMXMessage** of the **TsgcHTMX_Engine_Server**, checks the action name, and calls **ProcessTaskMoved** with the three values. That method looks up the task by its *Id*, applies

each date that parses (an unparsable date is simply left untouched), and then fires **OnTaskMoved**. The event always receives the raw strings sent by the browser, and it fires even when the id matches no task, so it is the right place to validate the request and to persist the new dates. Values coming from the browser are client-supplied, so check them before writing them to the database.

Member	Type	Default	Description
Draggable	Boolean	False	The user can drag a bar to move the task; the new dates are sent to the server.
Resizable	Boolean	False	The user can drag a bar edge to change its start or end date.
ProcessTaskMoved	method	–	Called by the host when a ganttTaskMoved message arrives: updates the task dates and fires OnTaskMoved.
OnTaskMoved	event	–	Fired after a drag or resize, with the task id and the new ISO dates.

```
// 1. enable the interaction
oGantt->Draggable = true;
oGantt->Resizable = true;
oGantt->OnTaskMoved = DoTaskMoved;

// 2. route the WebSocket message back into the component
void __fastcall TServer::DoHTMXMessage(TObject *Sender,
  TsgcWSConnection *aConnection, const UnicodeString aMessage,
  UnicodeString &aResponse)
{
  TStringList *oParams = new TStringList();
  try
  {
    UnicodeString vPath, vMethod;
    sgcHTMXParseMessage(aMessage, vPath, vMethod, oParams);
    if (SameText(oParams->Values["action"], "ganttTaskMoved"))
      oGantt->ProcessTaskMoved(oParams->Values["taskid"],
        oParams->Values["start"], oParams->Values["end"]);
  }
  __finally
  {
    delete oParams;
  }
}

// 3. persist the change
void __fastcall TServer::DoTaskMoved(TObject *Sender,
  const UnicodeString aTaskId, const UnicodeString aNewStart,
  const UnicodeString aNewEnd)
{
  qryUpdate->SQL->Text = "update TASKS set START_DATE = :s, END_DATE = :e where ID = :id";
  qryUpdate->ParamByName("s")->AsString = aNewStart;
  qryUpdate->ParamByName("e")->AsString = aNewEnd;
  qryUpdate->ParamByName("id")->AsString = aTaskId;
  qryUpdate->ExecSQL();
}
```

Loading from a Dataset

What it does: LoadFromDataSet fills the whole *Tasks* collection from a *TDataSet* in one call. You map one field to each part of a task: the id, the title, the start and end dates, and optionally the progress, the assignee and the com-

ma-separated dependency list. The method clears the current tasks, walks the dataset from the first record and adds one task per record, so the chart keeps the dataset order.

Nothing is loaded when the dataset is nil or not active, and nothing is loaded when the title field or the start field cannot be found. Records whose start field is null are skipped, and a record with no end date gets an end equal to its start. Every bar is created with the `hcPrimary` color. The field names are remembered, so a chart bound through **DataSource** with **DataAutoRefresh** enabled replays the same mapping and re-renders itself, pushing the new markup to the connected browsers, whenever the dataset changes.

Member	Type	Default	Description
LoadFrom-DataSet	method	–	Fills Tasks from a dataset, mapping one field to each part of a task.

```
qryTasks->Open();
oGantt->LoadFromDataSet(qryTasks, "ID", "TITLE", "START_DATE", "END_DATE",
  "PROGRESS", "ASSIGNEE", "DEPENDS_ON");
html = oGantt->HTML;
```

Output

The inherited **HTML** property returns the chart card: a header with the *Title*, a timeline scale, a label column with one task title (and assignee) per row, and a canvas holding the grid lines, the today marker, one positioned bar per task with its optional progress fill and resize handles, and the SVG layer with the dependency arrows. A footer with the range dates closes the card, and when the chart is *Draggable* or *Resizable* the hidden send form and the drag script are emitted too. **CSS** returns the layout styles. Serve these directly, or assign the HTML to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 11 properties available on the component.
- Methods — 4 public methods.
- Events — 1 published event.

TsgcHTMLComponent_PivotTable

TsgcHTMLComponent_PivotTable renders a cross-tab (pivot) table: rows are grouped by RowFields, columns by ColumnFields, and every cell aggregates one or more Measures.

Introduction

TsgcHTMLComponent_PivotTable turns a flat list of source rows into a cross-tab table. You declare which fields group the rows (down the left), which fields group the columns (across the top), and which values are aggregated inside every cell. The component then emits a finished HTML `<table>` through its inherited **HTML** property.

The aggregation is computed **server-side, in Delphi**. The browser never receives the source rows and never runs any pivot code: it receives a plain HTML table that is already grouped, aggregated, formatted and totalled. There is no JavaScript pivot engine, no client-side data set and no extra request.

Because the aggregation runs over the raw source rows, an average is the sum of the values divided by how many numeric values fall in the bucket, over *all* the source rows of the bucket. It is a true weighted average, never an "average of averages" of the sub-cells. Row totals, column totals and the grand total are each accumulated independently from the source rows for exactly the same reason, so a total is always exact and never an aggregation of aggregations.

Quick Start

1. Drop a **TsgcHTMLComponent_PivotTable** component on the form (or create it at runtime).

2. Add a **RowFields** item (`Region`), a **ColumnFields** item (`Quarter`) and a **Measures** item (source field `Amount`, aggregation `paSum`, format `#,##0.00`).

3. Fill it with **LoadFromDataSet** (or **AddRow**), then read the **HTML** property and serve it.

```
TsgcHTMLComponent_PivotTable *oPivot = new TsgcHTMLComponent_PivotTable(NULL);
try
{
  oPivot->RowFields->Add()->FieldName = "Region";
  oPivot->ColumnFields->Add()->FieldName = "Quarter";
  TsgcHTMLPivotMeasure *oMeasure = oPivot->Measures->Add();
  oMeasure->SourceField = "Amount";
  oMeasure->Caption = "Sales";
  oMeasure->Aggregation = paSum;
  oMeasure->Format = "#,##0.00";
  oPivot->LoadFromDataSet(qrySales);
  html = oPivot->HTML;
}
__finally
{
  delete oPivot;
}
```

Rows, Columns and Measures

What it does: *RowFields* lists the fields that group the source rows down the left of the table, *ColumnFields* lists the fields that group them across the top, and *Measures* lists the values aggregated inside every cell. Each collection may hold more than one item: several row fields produce several leading columns, several column fields pro-

duce a compound column header (the parts are joined with `/`), and several measures produce one sub-column per measure under every column group.

When *ColumnFields* is empty the table has no column groups: the measures are simply rendered as the last columns of every row.

Property	Type	Default	Description
RowFields	TsgcHTMLPivotFields	Empty collection	Fields that group the rows (down the left).
ColumnFields	TsgcHTMLPivotFields	Empty collection	Fields that group the columns (across the top).
Measures	TsgcHTMLPivotMeasures	Empty collection	Values aggregated inside every cell.

A **TsgcHTMLPivotField** (the item of *RowFields* and *ColumnFields*) exposes:

Member	Type	Description
FieldName	string	The name of the source field. It must match one of the *DataFields* names.
Caption	string	The header text. When empty, *FieldName* is used instead (this is what the read-only *DisplayCaption* returns).
DateGrouping	TsgcHTMLPivotDateGrouping	Groups the values of a date field: `pdgYear` (2026), `pdgQuarter` (Q1), `pdgMonth` (Jan to Dec), `pdgYearMonth` (2026-01) or `pdgYearQuarter` (2026 Q1). The values are read as ISO dates (`yyyy-mm-dd` , optionally followed by a time) whatever the locale. Values that are not a date fall in a `(blank)` member, shown last. The members always sort chronologically. Default `pdgNone` .
SortBy	TsgcHTMLPivotSortBy	The order of the members inside their parent group: `psbLabel` (by label), `psbValueAsc` or `psbValueDesc` (by the total of the member for the *SortMeasure*, inside its parent group). Ties and members without a value follow the label order. Default `psbLabel` .
SortMeasure	Integer	The zero based index of the measure whose totals order the members when *SortBy* sorts by value. An index out of range falls back to the label order. Default `0` .
TopN	Integer	Keeps the first N members of every parent group, after sorting, at every level. 0 keeps them all. Combine it with `psbValueDesc` to show the top N. Default `0` .
TopNOthers	Boolean	When False, the members beyond *TopN* are dropped and their rows are left out of the aggregation, totals included. When True, they are grouped in a single `Others` member, always shown last. Default `False` .
FilterValues	string	Comma separated member labels, trimmed and compared case insensitively. Only the source rows whose member is listed are aggregated. The

		labels are compared after *DateGrouping*, so `2025,2026` works on a year grouped date. Empty keeps every row. Default ''.
FilterExclude	Boolean	When True, the members listed in *FilterValues* are the ones left out. Default `False`.

A **TsgcHTMLPivotMeasure** (the item of *Measures*) exposes:

Member	Type	Description
SourceField	string	The name of the source field that is aggregated. It must match one of the *DataFields* names.
Caption	string	The header text of the measure. When empty, *SourceField* is used instead.
Aggregation	TsgcHTMLPivotAggregation	How the values are aggregated. Default `paSum`.
Format	string	A `FormatFloat` picture, for example `0.00` or `#,##0.##`. When empty the component uses `0.##`, or an integer when the aggregation is `paCount`.
ShowAs	TsgcHTMLPivotShowAs	How the value is displayed. Default `psaValue`. See *Show Values As* below.

Aggregation

What it does: every source row that shares the same row key and the same column key falls into the same bucket. The measure value of each of those rows is fed into the bucket, and the bucket is then resolved with the `TsgcHTMLPivotAggregation` of the measure.

Value	Description
paSum	The sum of the numeric values of the bucket. This is the default.
paCount	How many source rows fall in the bucket. Every row is counted, whether its measure value is a number or not.
paAverage	The sum of the numeric values divided by how many numeric values the bucket holds, computed over all the source rows of the bucket.
paMin	The smallest numeric value of the bucket.
paMax	The largest numeric value of the bucket.
paDistinctCount	The number of distinct non-empty values of the bucket. The values are trimmed and compared case sensitively. The result is an integer.

paMedian	The median of the numeric values of the bucket. With an even number of values, the mean of the two middle values.
paStdDev	The sample standard deviation (n-1) of the numeric values of the bucket. A bucket with fewer than two values renders *EmptyCellText*.

Numeric values. `paCount` counts every source row of the bucket. `paSum`, `paMin`, `paMax` and the denominator of `paAverage` only take into account the rows whose measure value is a number: empty and non-numeric values are ignored. A bucket that holds no numeric value at all renders *EmptyCellText*. The number parser is locale independent, it accepts an optional sign, digits and at most one decimal separator, which may be a dot or a comma.

Totals

What it does: *ShowRowTotals* adds a total column group at the right of every row, *ShowColumnTotals* adds a total row at the foot of the table, and *ShowGrandTotal* fills the cell where both meet. *TotalText* is the label used in both headers.

Each total is accumulated directly from the source rows, exactly like a cell, so an average total is the true average of all the rows it covers and not the average of the averages shown in the cells above it.

Property	Type	Default	Description
ShowRowTotals	Boolean	True	Adds a total group at the right of every row. It only appears when there is at least one column field and one measure.
ShowColumnTotals	Boolean	True	Adds a total row in the table footer.
ShowGrandTotal	Boolean	True	Fills the grand total cell where the row totals and the column totals meet.
TotalText	string	'Total'	The label of the total column and the total row.

Data

What it does: the source rows live in two public (runtime) string lists. *DataFields* holds the names of the source fields, and *Data* holds one line per source row, with the values separated by TAB characters in *DataFields* order. You rarely touch them directly.

There are three ways to fill the component:

Way	Description
AddRow	Fill *DataFields* yourself, then call **AddRow** once per source row. The values are positional: they follow the order of *DataFields*.

LoadFrom-DataSet	Call **LoadFromDataSet** with an open dataset. The component rebuilds *DataFields* from the field names used in *RowFields*, *ColumnFields* and *Measures*, then reads every record.
Data-Source	Assign the inherited *DataSource*. The component calls **LoadFromDataSet** with `DataSource.DataSet` , so the field list is again built automatically from *RowFields*, *Column-Fields* and *Measures*.

```
oPivot->DataFields->Add("Region");
oPivot->DataFields->Add("Quarter");
oPivot->DataFields->Add("Amount");
oPivot->AddRow(OPENARRAY(String, ("North", "Q1", "1200.50")));
oPivot->AddRow(OPENARRAY(String, ("North", "Q2", "980.00")));
oPivot->AddRow(OPENARRAY(String, ("South", "Q1", "1500.75")));
```

Appearance

What it does: the table is rendered with Bootstrap table classes. *Striped*, *Bordered*, *Hover* and *Compact* switch those classes on and off, *Caption* adds a `<caption>` above the table, *TableID* sets the id of the `<table>` element and *EmptyCellText* is what a bucket with no value renders. The inherited *CSSClass* and *Style* properties are appended to the generated class and style attributes.

Property	Type	Default	Description
Striped	Boolean	True	Alternates the background color of the rows (`table-striped`).
Bordered	Boolean	True	Draws a border around every cell (`table-bordered`).
Hover	Boolean	True	Highlights the row under the mouse pointer (`table-hover`).
Compact	Boolean	False	Reduces the cell padding (`table-sm`).
Caption	string	”	Optional caption rendered above the table.
TableID	string	”	The id of the generated <table> element.
EmptyCellText	string	‘-’	The text of a cell whose bucket has no value.

AI Query Bar

What it does: *AIQueryEnabled* renders a natural language query bar above the pivot. With an *AIAssistant* assigned, **ProcessAIQuery** turns the question into a validated layout and the rows, the columns, the measure and the aggregate become the RowFields, the ColumnFields and the single Measure. See AI Data Pack.

The layout you configured is snapshotted before the first question and restored by **ClearAIQuery**, so the pivot always comes back to the layout it was designed with.

Property	Type	Default	Description
AIQueryEnabled	Boolean	False	Renders the query bar. Off by default.
AIQueryPlace-holder	string	'Ask about your data...'	Placeholder of the query input.
AIQueryButton-Text	string	'Ask AI'	Caption of the ask button.
AIAssistant	TsgcHTMLA-IAssistant	nil	The assistant that answers the bar.
AIPersistState	Boolean	True	Keeps the query state in the request session when there is one, instead of in a hidden input of the bar.
QueryState	TsgcHTMLAI-QueryState	(empty)	Public. The question and the validated layout it produced.

Show Values As

What it does: the *ShowAs* member of a measure displays the aggregated value as a percent of the row, of the column or of the grand total, or as a running total across the columns.

Value	Description
psaValue	The aggregated value itself. This is the default.
psaPercentOfRow	The value divided by the row total of the same measure.
psaPercentOfColumn	The value divided by the column total of the same measure.
psaPercentOfGrandTo-tal	The value divided by the grand total of the same measure.
psaRunningTotal	The values accumulated across the columns of the row, in their display order.

A percent uses the *Format* of the measure (a `%` is added unless the picture already has one), or two decimals and `%` when *Format* is empty. A zero base renders *EmptyCellText*, and a total divided by itself shows 100%. With `psaRunningTotal`, the row total and the grand total show the plain value, and a column subtotal shows the running total at the end of its group.

Date Grouping, Sorting, Top N and Filters

What it does: every field of *RowFields*, *ColumnFields* and *FilterFields* can group ISO dates by year, quarter or month (*DateGrouping*), sort its members by label or by the total of a measure (*SortBy*, *SortMeasure*), keep only the

first N members of every parent group (*TopN*, with an optional `Others` member through *TopNOthers*) and keep or leave out a list of members (*FilterValues*, *FilterExclude*).

FilterFields holds fields that only filter: they add no row and no column. All the filters run on the source rows before the aggregation, so totals only include the rows that pass them. See RowFields for every member.

```
TsgcHTMLPivotField *oDate = oPivot->ColumnFields->Add();
oDate->FieldName = "OrderDate";
oDate->DateGrouping = pdgYearQuarter;
TsgcHTMLPivotField *oProduct = oPivot->RowFields->Add();
oProduct->FieldName = "Product";
oProduct->SortBy = psbValueDesc;
oProduct->TopN = 10;
oProduct->TopNOthers = true;
TsgcHTMLPivotField *oCountry = oPivot->FilterFields->Add();
oCountry->FieldName = "Country";
oCountry->FilterValues = "Spain,France";
```

Expandable Groups

What it does: with two or more row fields (or column fields), *Expandable* turns the outer levels into groups with subtotals and a toggle that expands or collapses their children in the browser. *ExpandAll* sets the initial state.

Property	Type	Default	Description
Expandable	Boolean	False	Turns the outer levels of the row and column fields into groups with subtotals that the user can expand and collapse.
ExpandAll	Boolean	True	The initial state of the groups when Expandable is True.
FilterFields	TsgcHTMLPivotFields	Empty collection	Fields that are neither rows nor columns, used only to filter the source rows.

Heatmap

What it does: colors the data cells of one measure with a background interpolated between two colors by value, with a readable text color. Totals are not colored.

Property	Type	Default	Description
Heatmap	Boolean	False	Colors the data cells of one measure by value, as a heatmap.
HeatmapMinColor	string	'#f8f9fa'	The heatmap color of the cell that holds the smallest value.
HeatmapMaxColor	string	'#0d6efd'	The heatmap color of the cell that holds the largest value.

HeatmapMeasure	Integer	0	The index of the measure whose cells the heatmap colors.

Drill-through and Export

What it does: with *DrillThroughURL*, clicking a data, subtotal or total cell posts its path (`pivot`, `rows` and `cols`) through htmx. The server calls **GetDrillThroughRows** to read the source rows behind the cell and answers the HTML shown in *DrillThroughTarget* or in a modal. *ShowExport* renders a CSV button, built in the browser, and an XLSX button that links to *ExportURL*, which the host serves with **SaveToXLSXStream**.

Property	Type	Default	Description
DrillThroughURL	string	''	The URL that the data, subtotal and total cells post to when the user clicks them, to show the source rows behind the cell.
DrillThroughTarget	string	''	Where the answer of a drill-through request is shown.
ShowExport	Boolean	False	Renders export buttons above the table.
ExportFileName	string	'pivot'	The file name of the CSV export.
ExportCSVText	string	'CSV'	The caption of the CSV export button.
ExportXLSXText	string	'XLSX'	The caption of the XLSX export button.
ExportURL	string	''	The URL of the XLSX export button.

Field Chooser and Linked Chart

What it does: *ShowFieldChooser* renders a drag and drop panel (Fields, Rows, Columns, Values) that posts every change to *FieldChooserURL*. The server calls **ApplyLayout** and answers the pivot HTML. *LinkedChart* fills a **TsgcHTMLComponent_Chart** from the pivot (one label per leaf row, one dataset per leaf column of *LinkedChartMeasure*) and renders it after the table.

Localization. The captions of the field chooser zones come from `sgcHTMLDefaultLocale`: *PivotFieldsText*, *PivotRowsText*, *PivotColumnsText* and *PivotValuesText* (English defaults Fields, Rows, Columns and Values). See Localization.

Property	Type	Default	Description
ShowFieldChooser	Boolean	False	Renders a drag and drop panel that lets the user lay the pivot out.

Field-ChooserURL	string	''	The URL the field chooser posts every lay-out change to.
LinkedChart	TsgcHTMLComponent_Chart	nil	A chart that is filled from the pivot and rendered after the table.
LinkedChart-Measure	Integer	0	The index of the measure whose values fill the LinkedChart.

Demo

The *Pivot lab* page (`/analytics/pivot`) of the Reports demo, `Demos\60.HTML\01.RunTime\15.Reports`, shows the field chooser, drill-through, CSV and XLSX export, heatmap, date grouping, top N and a linked chart working together.

Output

The inherited **HTML** property returns a `<div class="table-responsive">` wrapping a complete `<table>`: a `<thead>` with the column groups (and a second header row with the measure captions when there are column groups), a `<tbody>` with one row per row key, and a `<tfoot>` with the column totals. Every value is already aggregated and formatted, so no script is emitted and none is needed. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 40 properties available on the component.
- Methods — 10 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_TreeGrid

TsgcHTMLComponent_TreeGrid renders a hierarchical grid: the rows are a flat list of nodes, and the tree is derived from the Id and ParentId of each node.

Introduction

TsgcHTMLComponent_TreeGrid is a table whose rows can be nested. You never build a tree of objects: you add a flat list of **Nodes**, each one carrying an *Id* and the *ParentId* of the node it hangs from, and the component works out the hierarchy for you. The columns are declared in **Columns**, and each node's *Values* list holds one entry per column, in the order of that collection.

The table is rendered server-side, in Delphi, and returned through the inherited **HTML** property. The only client-side code is a small vanilla JavaScript block, emitted with the table, that expands and collapses a subtree when the user clicks a chevron. There is no round-trip to the server: the rows are already in the page, they are only shown or hidden.

Quick Start

1. Drop a **TsgcHTMLComponent_TreeGrid** component on the form (or create it at runtime).

2. Add two **Columns** (`Name` and `Amount`), then add the nodes with **AddNode**, giving each one an id and the id of its parent.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_TreeGrid *oTree = new TsgcHTMLComponent_TreeGrid(NULL);
try
{
  oTree->Columns->Add()->Caption = "Name";
  TsgcHTMLTreeGridColumn *oColumn = oTree->Columns->Add();
  oColumn->Caption = "Amount";
  oColumn->Align = tgaRight;
  oTree->AddNode("1", "", OPENARRAY(String, ("Europe", "15200.00")));
  oTree->AddNode("2", "1", OPENARRAY(String, ("Spain", "9800.00")));
  html = oTree->HTML;
}
__finally
{
  delete oTree;
}
```

Columns

What it does: *Columns* declares the columns of the grid, in the order they are rendered. The first *visible* column is the tree column: it carries the indent, the expand/collapse chevron and the node text. The other columns simply show the matching value of the node.

A **TsgcHTMLTreeGridColumn** exposes:

Member	Type	Default	Description
Caption	string	''	The header text. When empty, *FieldName* is shown instead.
FieldName	string	''	The name of the dataset field this column reads when the grid is filled with **LoadFromDataSet**.
Width	string	''	A CSS width for the column, for example `120px` or `20%`. When empty, the browser decides.
Align	TsgcHTMLTreeGridAlign	tgaLeft	Horizontal alignment: `tgaLeft`, `tgaCenter` or `tgaRight`.
Visible	Boolean	True	When False, the column is not rendered at all, neither in the header nor in the rows.

Nodes and Hierarchy

What it does: *Nodes* is a flat collection. The tree is not stored, it is derived every time the table is rendered, by matching the *ParentId* of each node against the *Id* of the others.

A **TsgcHTMLTreeGridNode** exposes:

Member	Type	Default	Description
Id	string	''	Identifies the node. Other nodes point at it through their *ParentId*.
ParentId	string	''	The *Id* of the parent node. Empty for a root.
Values	TStringList	Empty list	One value per column, in the order of the *Columns* collection. `ValueOf(aIndex)` returns the value of a column, or an empty string when the index is out of range.
Expanded	Boolean	True	Whether the children of this node are visible when the page is first rendered.

How the hierarchy is resolved. The rules are deliberately forgiving, so that imperfect data still renders something useful instead of losing rows:

Case	Result
ParentId is empty	The node is a root.

ParentId names an id that no node holds (an orphan)	The node becomes a root. It is rendered at the top level, it is never dropped.
ParentId equals the node's own Id	The node is treated as a root (self-parent guard).
Two nodes hold the same Id	The first node holding that id wins as the parent target. A later duplicate that would end up pointing at itself becomes a root.

Cycles. Corrupted data cannot hang the renderer or blow the stack. The guard works twice: a node is never rendered more than once, and the recursion stops at a maximum depth of 64 levels (`CS_SGC_TREEGRID_MAX_DEPTH`). Nodes deeper than that are skipped.

Binding to a Dataset

What it does: LoadFromDataSet fills the grid from a dataset. It needs three things: the dataset, the name of the field that holds the id, and the name of the field that holds the parent id.

When *Columns* is empty, the component creates one column per dataset field automatically, skipping the id field and the parent-id field, and right-aligning the numeric ones. When *Columns* already holds items, they are kept and each node reads its values through the *FieldName* of each column.

Assigning the inherited DataSource property is not enough for this component. Every other sgcHTML component can load itself from a *DataSource*, but a tree grid also needs the names of the id and the parent-id fields, and those cannot be guessed. The auto-load hook is therefore a no-op here: call **LoadFromDataSet** explicitly, passing the two field names.

```
// Columns is empty: one column per field is created automatically
oTree->LoadFromDataSet(qryOrgChart, "EmployeeID", "ManagerID");
```

Appearance

What it does: the table is rendered with Bootstrap table classes plus a small stylesheet of its own. *Striped*, *Bordered*, *Hover* and *ShowRowLines* control the look of the rows, *IndentPixels* how far a level is indented, and *ExpandedIcon* and *CollapsedIcon* what the chevron looks like. *Sortable* and *SortColumn* sort the siblings of every branch, and *TreeGridID* sets the id of the `<table>` element. The inherited *CSSClass* and *Style* properties are appended to the generated class and style attributes.

Property	Type	Default	Description
TreeGridID	string	''	The id of the generated <table> element. Empty means a unique id is generated on the first render.
ExpandedBy-Default	Boolean	True	Master switch for the initial expanded state.
IndentPixels	Integer	20	The width, in pixels, of one level of indentation.
Striped	Boolean	True	Alternates the background color of the rows.

Bordered	Boolean	False	Draws a border around every cell.
Hover	Boolean	True	Highlights the row under the mouse pointer.
ShowRow-Lines	Boolean	True	Draws a thin line under every row.
Sortable	Boolean	False	Sorts the siblings of every branch by the value of *SortColumn*.
SortColumn	Integer	0	The index of the column used to sort the siblings.
ExpandedIcon	string	'▼' (▼)	The chevron of an expanded node.
CollapsedIcon	string	'▶' (▶)	The chevron of a collapsed node.

A node is drawn expanded only when **both** its own *Expanded* property and the component's *ExpandedByDefault* property are True. Setting *ExpandedByDefault* to False therefore collapses the whole tree, whatever the individual nodes say.

ExpandedIcon and *CollapsedIcon* are honoured by the toggle as well, not only by the first render: the script emitted with the table writes back the icons this component is configured with, so a custom pair of icons is what the user keeps seeing while branches are opened and closed. Both icons are JavaScript encoded before they reach the script, so an icon holding markup, such as `<i class="bi bi-caret-down-fill"></i>`, reaches the browser intact and cannot close the `<script>` element.

TreeGridID is not hardcoded to a literal any more. Leave it empty and a unique element id is generated automatically the first time the component renders, and reused for the life of the instance, so two grids on the same page never collide and pushed fragments keep targeting the element this instance actually rendered. Set it explicitly if you need a stable, known id (for example to reach the element from your own JavaScript). The generated id is `sgcTreeGrid` plus a process-unique number, such as `sgcTreeGrid5`.

Output

The inherited **HTML** property returns a `<div class="table-responsive">` wrapping the `<table>`, followed by the expand/collapse script. Every node is a `<tr>` carrying `data-id`, `data-parent`, `data-depth` and, when it has children, `data-expanded`. Rows that start collapsed are emitted with `style="display:none;"`, so they are in the page from the start and the toggle only changes their visibility. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 13 properties available on the component.
- Methods — 3 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_AuditTrail

TsgcHTMLComponent_AuditTrail renders an audit log as an HTML table with a client-side filter, client-side paging and an optional Export button.

Introduction

TsgcHTMLComponent_AuditTrail is declared in the unit **sgcHTML_Component_ActivityFeed**, the same unit as **TsgcHTMLComponent_ActivityFeed**. Both components report activity, but they answer different questions: the activity feed is a chronological stream for humans to read, while the audit trail is a searchable record of who did what, to which resource, from which IP address, and with what result.

Each row of the table is a **TsgcHTMLAuditEntry** item in the **Entries** collection. The component renders the table server-side and returns the complete markup through its inherited **HTML** property, together with a small script that adds the filter box and the pager.

Quick Start

1. Drop a **TsgcHTMLComponent_AuditTrail** component on the form (or create it at runtime).

2. Add entries with **AddEntry**, and optionally turn the Export button on by setting **ShowExport** and **ExportURL**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_AuditTrail *oAudit = new TsgcHTMLComponent_AuditTrail(NULL);
try
{
  oAudit->Title = "Audit Trail";
  oAudit->AddEntry("ana", "login", "/admin", "10.0.0.7", "success");
  oAudit->AddEntry("ana", "update", "/admin/users/12", "10.0.0.7", "denied",
    "Role change rejected");
  oAudit->ShowExport = true;
  oAudit->ExportURL = "/audit/export.csv";
  html = oAudit->HTML;
}
__finally
{
  delete oAudit;
}
```

Audit Entries

What it does: *Entries* is a collection of **TsgcHTMLAuditEntry** items, one per table row. Add an entry at runtime with **AddEntry**, which appends the item and stamps *Timestamp* with the current date and time, or fill the collection at design time and set the timestamps yourself.

Property	Type	Description
Time-stamp	TDate-Time	When the action happened. Rendered with the *TimeFormat* mask. A zero timestamp renders as an empty cell.

User	string	Who performed the action.
Action	string	What was done, for example `login`, `update` or `delete`.
Resource	string	What the action was performed on, for example a path, a table or a record id.
IPAddress	string	The address the action came from.
Result	string	The outcome. The value drives the colour of a badge in the Result column (see below).
Details	string	Free text. The cell is truncated with an ellipsis and the full text is shown as a tooltip.

The *Result* value is matched case-insensitively to choose the badge colour: `success`, `ok`, `allowed` and `granted` render a green badge; `failed`, `fail`, `error`, `denied` and `rejected` render a red badge; any other non-empty value renders a grey badge. An empty *Result* renders no badge at all.

Filtering and Paging

What it does: *ShowFilter* adds a search box above the table and *PageSize* splits the rows into pages with a Prev / Next pager below it.

Both the filter and the pager are **entirely client-side**. All the rows are rendered into the page once, and the script the component emits simply shows and hides them: it compares the text of each row with the text you type, hides everything that does not match, and then displays only the slice that belongs to the current page. Typing in the filter box, or clicking Prev or Next, does **not** contact the server. There is no round-trip, no request and no re-render. This also means that every entry in the *Entries* collection is sent to the browser, so only load into the component the entries the user is allowed to see.

The filter matches against the whole text of a row, so typing a user name, an IP address or an action narrows the table in the same box. Editing the filter always returns to the first page. The pager is only rendered when there is at least one entry and *PageSize* is greater than zero; setting *PageSize* to 0 shows all the matching rows on a single page with no pager.

Property	Type	Default	Description
ShowFilter	Boolean	True	Shows the client-side filter box in the toolbar.
FilterPlaceholder	string	'Filter...'	Placeholder text of the filter box.
PageSize	Integer	25	Rows shown per page. 0 disables paging.

Export

What it does: *ShowExport* adds a button to the toolbar. The button is nothing more than a link (an `<a>` element with a `download` attribute) pointing at *ExportURL*.

The component does not generate the export file. It does not build a CSV, it does not build an XLSX, and it holds no export logic at all. When the user clicks the button the browser simply requests *ExportURL* from your serv-

er, and it is **your host code** that must answer that request with the file, in whatever format you choose, with the appropriate Content-Type. The component only renders the link.

The URL is sanitized before it is written into the markup: `javascript:` and `data:` URLs are rejected, so a value coming from configuration cannot turn the button into a script injection.

Property	Type	Default	Description
ShowExport	Boolean	False	Shows the Export button in the toolbar.
ExportURL	string	''	The URL the button links to. Your server serves the file.
ExportCaption	string	'Export'	Caption of the Export button.

Binding to a Dataset

What it does: LoadFromDataSet clears the *Entries* collection and walks the dataset from the first record to the last, mapping one field to each column. Every field-name parameter has a default equal to the column name, so a table whose columns are already called `Timestamp`, `User`, `Action`, `Resource`, `IPAddress`, `Result` and `Details` binds with a single call.

```
// Columns already named Timestamp, User, Action, Resource, IPAddress, Result, Details
oAudit->LoadFromDataSet(qryAudit);

// Different column names
oAudit->LoadFromDataSet(qryAudit, "LogDate", "UserName", "Operation",
  "Target", "ClientIP", "Status", "Notes");
```

A field that is not present in the dataset is skipped and leaves the corresponding entry property empty, and the whole call does nothing if the dataset is not assigned or not active.

Appearance

What it does: *Title* draws a heading above the toolbar, *Striped* and *Bordered* switch the Bootstrap table modifiers on and off, *TimeFormat* is the mask applied to the timestamp column, and *EmptyText* replaces the table when the collection is empty. *AuditID* is the id of the root element and the prefix of every generated element id.

Property	Type	Default	Description
Title	string	'Audit Trail'	Heading drawn above the toolbar. Empty hides the heading.
Striped	Boolean	True	Adds the striped-row table style.
Bordered	Boolean	False	Adds borders around every cell.
TimeFormat	string	'yyyy-mm-dd hh:nn:ss'	Format mask applied to the Timestamp column.

Empty-Text	string	'No audit entries'	Text shown instead of the table when there are no entries.
AuditID	string	''	Id of the root element. Leave it empty and a unique id is generated on the first render, so two audit trails on one page never collide.

Output

The inherited **HTML** property returns a `<div>` carrying the *AuditID* as its id, containing the optional title, the toolbar (filter box and Export link), a responsive `<table>` whose `<tbody>` holds one `<tr>` per entry, the Prev / Next pager, and the inline script that wires the filter and the pager. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

All the audit values are written as text, never as markup, and the client-side script only reads text and toggles row visibility, so an entry cannot inject script into the page.

Reference

- Properties — 13 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Form

TsgcHTMLComponent_Form builds a Bootstrap-styled HTML form from a collection of fields, with vertical, horizontal or inline layouts, submit/reset buttons, dataset binding and an optional AI-assisted "build from a description" bar.

Introduction

TsgcHTMLComponent_Form emits a complete `<form>` element. You describe each control as a **TsgcHTMLFormField** in the **Fields** collection (text, email, password, number, textarea, select, checkbox, radio, switch, file, date, time, color, range or hidden), choose a **Layout**, and the component assembles the Bootstrap markup, labels, help text, validation feedback and the submit/reset buttons. The finished markup is returned through the inherited **HTML** property.

Forms can be populated from a dataset with **LoadFromDataSet** (which creates one field per visible column) or filled with current row values through **LoadValuesFromDataSet**. When **AIBuildEnabled** is set, the form renders an extra input bar where the user types a plain-language description; the **OnAIBuildForm** event lets your code turn that description into a field definition.

Quick Start

1. Drop a **TsgcHTMLComponent_Form** component on the form (or create it at runtime).

2. Add one or more entries to the **Fields** collection, set the **Layout** and the **SubmitText**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Form *oForm = new TsgcHTMLComponent_Form(NULL);
try
{
  oForm->Action = "/save";
  oForm->Layout = flVertical;
  TsgcHTMLFormField *f1 = oForm->Fields->Add();
  f1->FieldType = ftText;
  f1->Name = "fullname";
  f1->Label_ = "Full Name";
  f1->Required = true;
  TsgcHTMLFormField *f2 = oForm->Fields->Add();
  f2->FieldType = ftEmail;
  f2->Name = "email";
  f2->Label_ = "Email";
  html = oForm->HTML;
}
__finally
{
  delete oForm;
}
```

Form Element and Layout

What it does: *Action* and *Method* set the form target and HTTP verb, while *FormID* and *CSSClass* annotate the `<form>` tag. *Layout* selects vertical, horizontal or inline arrangement; for the horizontal layout *LabelColWidth* and *FieldColWidth* split the 12-column grid between labels and inputs.

Property	Type	Default	Description
Action	string	''	The form action URL. Empty posts back to the current page.
Method	TsgcHTMLForm-Method	fmPost	HTTP method: fmGet or fmPost.
FormID	string	''	The id attribute of the <form> element.
Layout	TsgcHTMLForm-Layout	flVerti-cal	Field arrangement: flVertical, flHorizontal or flInline.
LabelCol-Width	Integer	3	Grid columns used by labels in the horizontal layout.
FieldCol-Width	Integer	9	Grid columns used by inputs in the horizontal layout.
CSSClass	string	''	Extra CSS class applied to the <form> element.

Fields

What it does: The **Fields** collection holds the form controls. Each item is a **TsgcHTMLFormField** exposing *Field-Type*, *Name*, *Label_*, *Placeholder*, *Value*, *HelpText*, *Required*, *Disabled*, *ReadOnly*, *Validation*, *Feedback*, *Options* (for select and radio), *FloatingLabel* and more. Use **LoadFromDataSet** to build the fields automatically from a dataset, or **LoadFieldsFromJSON** to build them from a JSON definition.

Property	Type	Default	Description
Fields	TsgcHTMLForm-Fields		Collection of TsgcHTMLFormField items rendered inside the form.

```
// Build the fields from a dataset, then prefill with the current row
oForm->LoadFromDataSet(qryCustomer);
oForm->LoadValuesFromDataSet(qryCustomer);
```

Buttons

What it does: *SubmitText* and *SubmitStyle* (or *SubmitClass*) control the submit button. Set *ShowReset* to add a re-set button labelled by *ResetText*.

Property	Type	Default	Description

SubmitText	string	locale “Submit”	Caption of the submit button.
SubmitClass	string	”	Explicit CSS class for the submit button. Overrides SubmitStyle when set.
SubmitStyle	TsgcHTMLButtonStyle	bsPrimary	Bootstrap button style used when SubmitClass is empty.
ShowReset	Boolean	False	Adds a reset button next to the submit button.
ResetText	string	locale “Reset”	Caption of the reset button.

AI Build

What it does: When *AIBuildEnabled* is True the form renders an extra bar (placeholder text from *AIBuildPlaceholder*) where the user describes the form in plain language. Call **BuildFromAIDescription** with that text; the **OnAIBuildForm** event hands you the description and lets you return a JSON field definition, which the component loads through **LoadFieldsFromJSON**.

Property	Type	Default	Description
AIBuildEnabled	Boolean	False	Shows the AI "build from a description" input bar above the form.
AIBuildPlaceholder	string	‘Describe your form…’	Placeholder text shown in the AI build input.

Smart Paste

What it does: *SmartPasteEnabled* renders a paste zone above the form. What a human pastes there is posted as the `formSmartPaste` action, with the FormID in the `form` field and the text, line breaks included, in the `text` field. **ProcessSmartPaste** hands it to the *AIAssistant* together with the field whitelist, which is the name, the label and the type of every named field and no value of the form, and writes what comes back into the fields that exist.

A field that was filled that way is marked as AI filled: its label carries a small badge and the control the valid styling, so a human sees what was written. **ClearAIFilled** drops the marks and **LoadValuesFromJSON** does the same from JSON your own code produced. Nothing is ever emitted as markup: a value goes into the Value of the field, which the renderer escapes. See AI Data Pack.

Property	Type	Default	Description
SmartPasteEnabled	Boolean	False	Renders the paste zone. Off by default, and then nothing is rendered.
SmartPasteHint	string	”	Label of the zone. Empty uses the localized text.

AIAssistant	TsgcHTMLAIAssistant	nil	The assistant that reads the pasted text.

Output

The inherited **HTML** property returns the optional AI build bar followed by the `<form>` element with its fields and buttons. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 18 properties available on the component.
- Methods — 7 public methods.
- Events — 1 published event.

TsgcHTMLComponent_Edit

TsgcHTMLComponent_Edit renders a single-line text input with an optional label and help text, supporting every HTML5 input type and binding to a dataset field.

Introduction

TsgcHTMLComponent_Edit emits a Bootstrap-styled `<input>` control wrapped in a `<div class="mb-3">`. You set a label, a placeholder and an initial value, pick the input type (text, email, password, number and so on through the **InputType** property) and the component builds the markup for you, returning the complete HTML through its inherited **HTML** property.

An optional **HelpText** line is rendered below the field, and the control can be made **Required**, **Disabled** or **ReadOnly**. Bind **DataField** (together with the inherited **DataSource**) to load the initial value straight from a dataset record.

Quick Start

1. Drop a **TsgcHTMLComponent_Edit** component on the form (or create it at runtime).

2. Set the **Label_**, **Name** and **Placeholder**, and choose an **InputType** if it is not plain text.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Edit *oEdit = new TsgcHTMLComponent_Edit(NULL);
try
{
  oEdit->Label_ = "Email address";
  oEdit->Name = "email";
  oEdit->InputType = itEmail;
  oEdit->Placeholder = "you@example.com";
  oEdit->Required = true;
  oEdit->HelpText = "We never share your email.";
  html = oEdit->HTML;
}
__finally
{
  delete oEdit;
}
```

Field Identity and Type

What it does: *Name* sets the form field name, *Label_* the visible caption and *EditID* the element id (a value is generated from the name when left blank). *InputType* chooses the HTML5 input type, *Value* the initial content and *Placeholder* the greyed-out hint text.

Property	Type	Default	Description
Name	string	''	The HTML name attribute used when the form is submitted.
Label_	string	''	Caption rendered in a <label> above the input. Empty hides the label.
InputType	TsgcHTMLInputType	itText	HTML5 input type: itText, itEmail, itPassword, itNumber, itTel, itURL, itSearch, itDate, itTime, itDateTimeLocal, itMonth, itWeek, itColor, itRange, itFile or itHidden.
Value	string	''	Initial value placed in the input.
Placeholder	string	''	Hint text shown while the field is empty.
EditID	string	''	The id of the generated <input>. When empty an id of edit_<Name> is generated.
HelpText	string	''	Small help line drawn below the input. Empty hides it.

```
oEdit->InputType = itPassword;
oEdit->EditID = "pwd";
oEdit->Value = "";
```

State and Validation

What it does: *Required* marks the field as mandatory, *Disabled* greys it out and excludes it from submission, and *ReadOnly* shows the value but prevents editing.

Property	Type	Default	Description
Required	Boolean	False	Adds the required attribute so the browser blocks empty submission.
Disabled	Boolean	False	Disables the input; it is greyed out and not submitted.
ReadOnly	Boolean	False	Shows the value but prevents the user from editing it.

Dataset Binding

What it does: Set *DataField* to the name of a field and assign the inherited *DataSource*; when the component loads from the data source the input's value is taken from that field of the current record.

```
oEdit->DataField = "CustomerName";
oEdit->DataSource = dsCustomers;
```

Output

The inherited **HTML** property returns a `<div class="mb-3">` wrapping the optional `<label>`, the `<input>` and the optional help `<div>`. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 11 properties available on the component.
- Methods — this component has no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_Memo

TsgcHTMLComponent_Memo renders a multi-line textarea with an optional label, a configurable number of rows, and binding to a dataset field.

Introduction

TsgcHTMLComponent_Memo emits a Bootstrap-styled `<textarea>` control wrapped in a `<div class="mb-3">`. You set a label, a placeholder and the initial text, choose how many visible rows the box has through the **Rows** property, and the component builds the markup for you, returning the complete HTML through its inherited **HTML** property.

The control can be made **Required**, **Disabled** or **ReadOnly**. Bind **DataField** (together with the inherited **DataSource**) to load the initial text straight from a dataset record. Unlike the single-line edit, the memo has no help-text line; its content is the textarea body rather than a value attribute.

Quick Start

1. Drop a **TsgcHTMLComponent_Memo** component on the form (or create it at runtime).

2. Set the **Label_**, **Name**, **Placeholder** and the number of **Rows**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Memo *oMemo = new TsgcHTMLComponent_Memo(NULL);
try
{
  oMemo->Label_ = "Comments";
  oMemo->Name = "comments";
  oMemo->Rows = 6;
  oMemo->Placeholder = "Tell us what you think...";
  html = oMemo->HTML;
}
__finally
{
  delete oMemo;
}
```

Field Identity and Size

What it does: *Name* sets the form field name, *Label_* the visible caption and *MemoID* the element id (a value is generated from the name when left blank). *Rows* sets the visible height of the textarea, *Value* the initial text and *Placeholder* the greyed-out hint.

Property	Type	Default	Description
Name	string	''	The HTML name attribute used when the form is submitted.
Label_	string	''	Caption rendered in a <label> above the textarea. Empty hides the label.

Rows	Inte-ger	4	Number of visible text rows (the textarea rows attribute).
Value	string	"	Initial text placed inside the textarea.
Placehold-er	string	"	Hint text shown while the textarea is empty.
MemoID	string	"	The id of the generated <textarea>. When empty an id of memo_<Name> is generated.

```
oMemo->MemoID = "notes";
oMemo->Rows = 10;
oMemo->Value = "Initial text";
```

State and Validation

What it does: *Required* marks the field as mandatory, *Disabled* greys it out and excludes it from submission, and *ReadOnly* shows the text but prevents editing.

Property	Type	Default	Description
Required	Boolean	False	Adds the required attribute so the browser blocks empty submis-sion.
Disabled	Boolean	False	Disables the textarea; it is greyed out and not submitted.
ReadOnly	Boolean	False	Shows the text but prevents the user from editing it.

Dataset Binding

What it does: Set *DataField* to the name of a field and assign the inherited *DataSource*; when the component loads from the data source the textarea's text is taken from that field of the current record.

```
oMemo->DataField = "Notes";
oMemo->DataSource = dsCustomers;
```

Output

The inherited **HTML** property returns a `<div class="mb-3">` wrapping the optional `<label>` and the `<textarea>`. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 10 properties available on the component.
- Methods — this component has no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_CheckBox

TsgcHTMLComponent_CheckBox renders a single checkbox (or a toggle switch) with a label, an initial checked state, and binding to a boolean dataset field.

Introduction

TsgcHTMLComponent_CheckBox emits a Bootstrap-styled checkbox wrapped in a `<div class="form-check">`. You set a label and the initial **Checked** state, and the component builds the markup for you, returning the complete HTML through its inherited **HTML** property. Set **Switch** to render the same control as a sliding toggle switch instead of a square box.

The control can be made **Disabled**. Bind **DataField** (together with the inherited **DataSource**) to set the initial checked state from a boolean field of a dataset record. Unlike the text inputs, the checkbox has no Value, Placeholder or help-text property; its state is simply checked or unchecked.

Quick Start

1. Drop a **TsgcHTMLComponent_CheckBox** component on the form (or create it at runtime).

2. Set the **Label_**, **Name** and the initial **Checked** state; set **Switch** for a toggle look.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_CheckBox *oCheck = new TsgcHTMLComponent_CheckBox(NULL);
try
{
  oCheck->Label_ = "Subscribe to newsletter";
  oCheck->Name = "subscribe";
  oCheck->Checked = true;
  oCheck->Switch = true;
  html = oCheck->HTML;
}
__finally
{
  delete oCheck;
}
```

Identity and State

What it does: *Name* sets the form field name, *Label_* the visible caption and *CheckBoxID* the element id (a value is generated from the name when left blank). *Checked* sets the initial state, *Switch* renders the control as a toggle switch, and *Disabled* greys it out.

Property	Type	Default	Description
Name	string	''	The HTML name attribute used when the form is submitted.
Label_	string	''	Caption rendered in the <label> next to the checkbox.

Checked	Boolean	False	Initial checked state of the box.
Switch	Boolean	False	Renders the control as a Bootstrap toggle switch instead of a square box.
Disabled	Boolean	False	Disables the checkbox; it is greyed out and not submitted.
Check-BoxID	string	”	The id of the generated <input>. When empty an id of chk_<Name> is generated.

```
oCheck->CheckBoxID = "optIn";
oCheck->Switch = false;
oCheck->Disabled = false;
```

Dataset Binding

What it does: Set *DataField* to the name of a boolean field and assign the inherited *DataSource*; when the component loads from the data source the checked state is taken from that field of the current record.

```
oCheck->DataField = "IsActive";
oCheck->DataSource = dsCustomers;
```

Output

The inherited **HTML** property returns a `<div class="form-check mb-3">` (with `form-switch` added when **Switch** is True) wrapping the `<input type="checkbox">` and its `<label>`. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 7 properties available on the component.
- Methods — this component has no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_RadioGroup

TsgcHTMLComponent_RadioGroup renders a set of mutually exclusive radio buttons from a list of options, with a group label, a pre-selected item, and an optional inline layout.

Introduction

TsgcHTMLComponent_RadioGroup emits a group of Bootstrap-styled radio buttons that share one `name` so only one can be selected at a time. You fill the **Items** string list with the option captions, optionally pre-select one with **SelectedIndex**, and the component builds the markup for you, returning the complete HTML through its inherited **HTML** property.

Each item's caption is used both as the radio `value` and as its visible label. Set **InlineLayout** to lay the buttons out horizontally, and **Disabled** to grey the whole group out.

Quick Start

1. Drop a **TsgcHTMLComponent_RadioGroup** component on the form (or create it at runtime).

2. Set the **Label_** and **Name**, add the option captions to **Items**, and set **SelectedIndex** to pre-select one.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_RadioGroup *oRadio = new TsgcHTMLComponent_RadioGroup(NULL);
try
{
  oRadio->Label_ = "Shipping method";
  oRadio->Name = "shipping";
  oRadio->Items->Add("Standard");
  oRadio->Items->Add("Express");
  oRadio->Items->Add("Overnight");
  oRadio->SelectedIndex = 0;
  html = oRadio->HTML;
}
__finally
{
  delete oRadio;
}
```

Group Identity and Options

What it does: *Name* sets the shared form field name (every button in the group uses it), *Label_* the group caption and *RadioGroupID* the base id used to derive each item's id. *Items* holds the option captions and *SelectedIndex* picks which one is selected initially.

Property	Type	Default	Description
Name	string	''	The shared HTML name attribute submitted for the selected button.

Label_	string	''	Caption rendered in a <label> above the group. Empty hides it.
Items	TStringList	(empty)	The option captions; each is used as both the radio value and its label.
SelectedIndex	Integer	-1	Zero-based index of the pre-selected item; -1 selects none.
RadioGroupID	string	''	Base id for the group. When empty rg_<Name> is used; each item id is <base>_<index>.

```
oRadio->Items->Clear();
oRadio->Items->Add("Yes");
oRadio->Items->Add("No");
oRadio->SelectedIndex = 1;
```

Layout and State

What it does: *InlineLayout* lays the buttons out side by side instead of stacked, and *Disabled* greys the whole group out so none of the buttons can be selected.

Property	Type	Default	Description
InlineLayout	Boolean	False	Lays the radio buttons out horizontally (form-check-inline).
Disabled	Boolean	False	Disables every button in the group; none can be selected and none is submitted.

```
oRadio->InlineLayout = true;
oRadio->Disabled = false;
```

Output

The inherited **HTML** property returns a `<div class="mb-3">` wrapping the optional group `<label>` and one `<div class="form-check">` per item, each holding an `<input type="radio">` and its label. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 7 properties available on the component.
- Methods — this component has no public methods of its own.

- Events — this component has no published events.

TsgcHTMLComponent_Select

TsgcHTMLComponent_Select renders a Bootstrap-styled HTML <select> drop-down from a collection of options, with optional option groups, single or multiple selection, sizing and dataset binding.

Introduction

TsgcHTMLComponent_Select emits a labelled `<select>` element. You fill the **Options** collection with value/text pairs (optionally grouped with `<optgroup>`), choose single or multiple selection, and the component builds the Bootstrap markup. The finished markup is returned through the inherited **HTML** property.

Options can be added one at a time with **AddOption** and **AddOptionGroup**, or loaded from a dataset with **LoadFromDataSet** by naming the value, text and (optionally) group columns.

Quick Start

1. Drop a **TsgcHTMLComponent_Select** component on the form (or create it at runtime).

2. Set the **Name** and **Label_**, then add entries with **AddOption**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Select *oSelect = new TsgcHTMLComponent_Select(NULL);
try
{
  oSelect->Name = "country";
  oSelect->Label_ = "Country";
  oSelect->Placeholder = "Choose...";
  oSelect->AddOption("us", "United States");
  oSelect->AddOption("uk", "United Kingdom", true);
  html = oSelect->HTML;
}
__finally
{
  delete oSelect;
}
```

Element and Behaviour

What it does: *Name* and *SelectID* identify the control, *Label_* draws a label above it and *Placeholder* adds a disabled first option. *Multiple* turns it into a multi-select, *Size* picks the Bootstrap height, *VisibleItems* sets how many rows show at once, and *Required* / *Disabled* set the matching attributes.

Property	Type	Default	Description
Name	string	''	The name attribute submitted with the form.
Label_	string	''	Optional label drawn above the select.

Placeholder	string	''	Disabled, pre-selected first option used as a prompt.
SelectID	string	''	The id of the <select> element. Defaults to 'sel_' + Name.
Multiple	Boolean	False	Allows selecting several options at once.
Size	TsgcHTMLSelectSize	ssDefault	Control height: ssDefault, ssSmall or ssLarge.
VisibleItems	Integer	0	Number of rows shown at once (the size attribute). 0 leaves it as a drop-down.
Required	Boolean	False	Marks the select as required.
Disabled	Boolean	False	Disables the select.

Options

What it does: The **Options** collection holds the choices. Each item is a **TsgcHTMLSelectOption** exposing *Value*, *Text*, *Selected*, *Disabled* and *Group*. Add options with **AddOption**, grouped options with **AddOptionGroup**, or load them from a dataset with **LoadFromDataSet**.

Property	Type	Default	Description
Options	TsgcHTMLSelectOptions		Collection of TsgcHTMLSelectOption items rendered as <option> elements.

```
// Grouped options
oSelect->AddOptionGroup("Europe", "uk", "United Kingdom");
oSelect->AddOptionGroup("Europe", "fr", "France");
oSelect->AddOptionGroup("America", "us", "United States");
```

Output

The inherited **HTML** property returns a `<div>` wrapping the optional label and the `<select>` element with its options and groups. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 10 properties available on the component.

- Methods — 3 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_InputGroup

TsgcHTMLComponent_InputGroup renders a Bootstrap input group: a text input with optional prepended and appended add-ons such as currency symbols, units or icons.

Introduction

TsgcHTMLComponent_InputGroup emits a Bootstrap `input-group`: a single `<input>` wrapped with optional add-ons before (**PrependText**) and after (**AppendText**) it. Add-ons are handy for prefixes like `$` or `@` and suffixes like `.00` or `kg`. The finished markup is returned through the inherited **HTML** property.

You can drop the component on a form and set its properties, or build a group in one line without an instance using the static **Build** class function.

Quick Start

1. Drop a **TsgcHTMLComponent_InputGroup** component on the form (or create it at runtime).

2. Set **InputName**, then a **PrependText** and/or **AppendText**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_InputGroup *oGroup = new TsgcHTMLComponent_InputGroup(NULL);
try
{
  oGroup->PrependText = "$";
  oGroup->InputName = "amount";
  oGroup->Placeholder = "0.00";
  oGroup->AppendText = ".00";
  html = oGroup->HTML;
}
__finally
{
  delete oGroup;
}
```

Add-ons and Input

What it does: *PrependText* and *AppendText* are the add-ons shown before and after the field. *InputName*, *InputValue* and *Placeholder* configure the field itself; *InputType* (a string) or *InputTypeEnum* (a typed enum) set the HTML input type, *Size* picks the control height and *GroupID* annotates the wrapper.

Property	Type	Default	Description
PrependText	string	''	Add-on text shown before the input.
AppendText	string	''	Add-on text shown after the input.
InputName	string	''	The name attribute submitted with the form.

InputValue	string	''	Initial value of the input.
Placeholder	string	''	Placeholder text shown when the input is empty.
InputType	string	'text'	The HTML input type as a raw string (e.g. text, email, number).
InputType-Enum	TsgcHTM-LInputType	itText	The input type as a typed enum, used when InputType is empty.
Size	TsgcHTM-LInput-GroupSize	igsDe-fault	Control height: igsDefault, igsSmall or igsLarge.
GroupID	string	''	The id of the input-group wrapper .

Build helper

What it does: The static **Build** class function returns the input-group markup in a single call, without you having to create, configure and free an instance. Two overloads are provided: one takes the input type as a `TsgcHTMLInput-Type` enum, the other defaults it to text.

```
// One-liner, no instance to manage
String vHTML = TsgcHTMLComponent_InputGroup::Build("@", "username", "Enter username");
```

Output

The inherited **HTML** property returns a `<div class="input-group">` wrapping the optional prepend add-on, the `<in-put>` and the optional append add-on. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 9 properties available on the component.
- Methods — 1 public method (with 2 overloads).
- Events — this component has no published events.

TsgcHTMLComponent_AutoComplete

TsgcHTMLComponent_AutoComplete renders a text input with suggestions, either as a static HTML <datalist> filled from Items, or as a server-driven search where every keystroke is debounced in the browser, sent over the WebSocket and answered by the OnSearch event.

Introduction

TsgcHTMLComponent_AutoComplete emits a labelled text `<input>` with a suggestion list, and it works in two modes, selected with the **Mode** property.

In **acDatalist** mode (the default) the input is linked to a native `<datalist>` built from the **Items** string list. All the suggestions travel with the page, the browser does the matching, and no JavaScript and no server round-trip are involved.

In **acServer** mode the input is server-driven. The component emits a small script that debounces the typing (**DebounceMs**) and, once at least **MinLength** characters have been typed, sends the query over the WebSocket connection the page already has open, as `{"action":"autoCompleteSearch","autocomplete":"<AutoCompleteID>","query":"<text>"}`. On the server you call **ProcessSearch**, which fires **OnSearch**; your handler fills the results, and the string returned by **GetResultsFragmentHTML** is pushed back to the browser as an out-of-band htmx fragment that replaces the suggestions panel. The search itself always runs on the server, so the data never has to be shipped to the client.

In both modes the finished markup is returned through the inherited **HTML** property.

Quick Start

1. Drop a **TsgcHTMLComponent_AutoComplete** component on the form (or create it at runtime).

2. Set the **Name** and **Label_**, then fill the **Items** list with suggestions (this is the default **acDatalist** mode).

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_AutoComplete *oAuto = new TsgcHTMLComponent_AutoComplete(NULL);
try
{
  oAuto->Name = "city";
  oAuto->Label_ = "City";
  oAuto->Placeholder = "Start typing...";
  oAuto->Items->Add("London");
  oAuto->Items->Add("Lisbon");
  oAuto->Items->Add("Lima");
  html = oAuto->HTML;
}
__finally
{
  delete oAuto;
}
```

Modes

What it does: *Mode* selects how the suggestions are produced. *acDatalist* (the default) renders the input plus a native `<datalist>` holding one `<option>` per entry of *Items*, and the browser matches locally. *acServer* renders the input plus a hidden transport form, an empty suggestions panel and a small debounce script, and every query is answered by the server through **OnSearch**. When the mode is *acServer*, *Items* is not rendered at all.

Property	Type	Default	Description
Mode	TsgcHTMLAutoCompleteMode	acDatalist	acDatalist renders a static <datalist> from Items; acServer sends every debounced keystroke to the server.

Input and Suggestions

What it does: *Name* and *AutoCompleteID* identify the control, *Label_* draws a label above it, and *Placeholder* and *Value* configure the field. *Items* holds the suggestion list rendered into the `<datalist>` in *acDatalist* mode, and can also be filled from a dataset with **LoadFromDataSet**. *MinLength* is the minimum number of characters before suggestions are offered; in *acServer* mode it is enforced by the emitted script, which sends no query until the input holds that many characters.

Property	Type	Default	Description
Name	string	''	The name attribute submitted with the form.
Label_	string	''	Optional label drawn above the input.
Placeholder	string	''	Placeholder text shown when the input is empty.
Value	string	''	Initial value of the input.
Items	TStringList	(empty)	The suggestions rendered into the <datalist> (acDatalist mode).
AutoCompleteID	string	'sgcAutoComplete'	The id of the input and the base for the datalist, form and results panel ids.
MinLength	Integer	1	Minimum number of characters before suggestions are offered.

Server-Driven Search

What it does: When *Mode* is *acServer*, the browser waits *DebounceMs* milliseconds after the last keystroke and then sends the query over the WebSocket. While the round-trip is in flight the panel shows *LoadingText*. On the server, **ProcessSearch** fires the **OnSearch** event, where you fill the results (one suggestion per line); **GetResultsFragmentHTML** then returns the out-of-band fragment that replaces the suggestions panel on the client. *MaxResults* caps how many suggestions are rendered, *HighlightMatch* wraps the matched substring of each suggestion in a `<mark>` element, and *NoResultsText* is shown when the search returned nothing.

Property	Type	Default	Description
DebounceMs	Integer	250	Milliseconds the browser waits after the last keystroke before sending the query.
MaxResults	Integer	10	Maximum number of suggestions rendered in the panel.
Highlight-Match	Boolean	True	Wraps the matched substring of each suggestion in a <mark> element.
NoResults-Text	string	'No re-sults'	Shown in the suggestions panel when the search returned nothing.
LoadingText	string	'Search-ing...'	Shown in the suggestions panel while the round-trip is in flight.

The round-trip has four steps.

1. The page is rendered with **Mode** set to *acServer* and an **OnSearch** handler assigned.

2. The user types. After **DebounceMs** milliseconds of silence, and only if the input holds at least **MinLength** characters, the browser sends `{"action":"autoCompleteSearch","autocomplete":"<AutoCompleteID>","query":"<text>"}` over the WebSocket and shows **LoadingText**.

3. The host receives the message, recognises the `autoCompleteSearch` action and calls **ProcessSearch** with the query. That fires **OnSearch**, and the handler fills `aResults` with the matching suggestions, one per line.

4. The host pushes **GetResultsFragmentHTML** back to the calling browser with **TsgcHTMX_Engine_Server.PushFragment**. The fragment targets the suggestions panel (`<AutoCompleteID>_results`) and replaces its content out of band, so nothing else on the page is touched. Clicking a suggestion copies it into the input and closes the panel.

```
// 1. server driven input
oAuto = new TsgcHTMLComponent_AutoComplete(NULL);
oAuto->AutoCompleteID = "city_ac";
oAuto->Name = "city";
oAuto->Label_ = "City";
oAuto->Mode = acServer;
oAuto->MinLength = 2;
oAuto->DebounceMs = 200;
oAuto->MaxResults = 8;
oAuto->HighlightMatch = true;
oAuto->OnSearch = DoAutoCompleteSearch;

// 2. the search runs on the server: one suggestion per line
void __fastcall TForm1::DoAutoCompleteSearch(TObject *Sender,
  const UnicodeString aQuery, UnicodeString &aResults)
{
  aResults = "";
  for (int i = 0; i < FCities->Count; i++)
    if (FCities->Strings[i].LowerCase().Pos(aQuery.LowerCase()) > 0)
      aResults = aResults + FCities->Strings[i] + sLineBreak;
}

// 3. and 4. the inbound message: search and push the results panel back
void __fastcall TForm1::HTMXMessage(TObject *Sender,
  TsgcWSConnection *aConnection, const UnicodeString aMessage,
  UnicodeString &aResponse)
{
  TStringList *oParams = new TStringList();
  try
  {
    UnicodeString vPath, vMethod;
    sgcHTMXParseMessage(aMessage, vPath, vMethod, oParams);
    if (SameText(oParams->Values["action"], "autoCompleteSearch"))
    {
      oAuto->ProcessSearch(oParams->Values["query"]);
      oHTMX->PushFragment(aConnection->Guid, oAuto->GetResultsFragmentHTML());
```

```
      }
    }
    __finally
    {
      delete oParams;
    }
  }
```

The results your handler produced stay available on the read-only **Results** string list, and the query that produced them on **LastQuery** (the highlight is computed against it). **GetResultsHTML** returns the inner markup of the panel on its own, when you want to embed it instead of pushing it.

Output

In **acDatalist** mode the inherited **HTML** property returns a `<div>` wrapping the optional label, the `<input>` (linked to the datalist with its `list` attribute) and a `<datalist>` holding one `<option>` per entry in **Items**.

In **acServer** mode it returns a `<div>` wrapping a hidden form that transports the query over the WebSocket, the `<input>` (with `autocomplete="off"`), an empty suggestions panel with id `<AutoCompleteID>_results`, and the debounce script. The panel is filled later by the fragment returned from **GetResultsFragmentHTML**.

In both modes you serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 13 properties available on the component.
- Methods — 4 public methods to load the items, run the search and build the results panel.
- Events — 1 event fired when a server-driven search arrives.

TsgcHTMLComponent_DatePicker

TsgcHTMLComponent_DatePicker renders a Bootstrap-styled date, time or date-time input field, with optional label, placeholder, value and minimum / maximum range.

Introduction

TsgcHTMLComponent_DatePicker emits an HTML `<input>` element wrapped in a Bootstrap form group. The **Mode** property chooses between a date, a time or a combined date-time field by setting the native HTML input `type` (`date`, `time` or `datetime-local`), so the browser's own picker is used. The component builds the markup for you and returns it through its inherited **HTML** property.

No external library is required: the picker is the browser's built-in control, styled with the Bootstrap `form-control` class.

Quick Start

1. Drop a **TsgcHTMLComponent_DatePicker** component on the form (or create it at runtime).

2. Set the **Name**, choose a **Mode** and optionally a **Label_** and **Value**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_DatePicker *oPicker = new TsgcHTMLComponent_DatePicker(NULL);
try
{
  oPicker->Name = "start_date";
  oPicker->Mode = dmDate;
  oPicker->Label_ = "Start date";
  oPicker->Value = "2026-01-01";
  html = oPicker->HTML;
}
__finally
{
  delete oPicker;
}
```

Mode and Field

What it does: *Mode* selects the kind of field (date, time or date-time). *Name* sets the form field name that the value is submitted under, *DatePickerID* sets the element id (defaulting to `dp_` plus the name), and *Label_* draws an associated label above the input.

Property	Type	Default	Description
Mode	TsgcHTMLDatePicker-Mode	dmDate	Field kind: dmDate, dmTime or dmDateTime.
Name	string	''	Form field name the value is submitted under.

DatePick-erID	string	”	The id of the generated <input>. When empty, `dp_` plus the Name is used.
Label_	string	”	Optional label drawn above the input. Empty hides the label.

```
oPicker->Mode = dmDateTime;
oPicker->Name = "appointment";
oPicker->Label_ = "Appointment";
```

Value and Constraints

What it does: *Value* sets the initial value (in the format the chosen field expects). *Placeholder* supplies hint text, *MinDate* and *MaxDate* restrict the selectable range, and *Required* marks the field as required for form validation.

Property	Type	Default	Description
Value	string	”	Initial value, in the format the chosen field expects (for example `2026-01-31`).
Placehold-er	string	‘Select date...’	Hint text shown when the field is empty.
MinDate	string	”	Earliest selectable value (sets the input `min` attribute).
MaxDate	string	”	Latest selectable value (sets the input `max` attribute).
Required	Boolean	False	When True the field is marked required for HTML form validation.

```
oPicker->MinDate = "2026-01-01";
oPicker->MaxDate = "2026-12-31";
oPicker->Required = true;
```

One-Line Build

What it does: The static **Build** class function returns the field markup in a single call, without creating and freeing an instance yourself. Pass a name, an optional mode, an optional label and an optional value.

```
html = TsgcHTMLComponent_DatePicker::Build("start_date", dmDate, "Start date", "2026-01-01");
```

Output

The inherited **HTML** property returns a `<div class="mb-3">` wrapping the optional `<label>` and the `<input>` element. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 9 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_FileUpload

TsgcHTMLComponent_FileUpload renders a Bootstrap file-upload form with an optional drag-and-drop drop zone, icon, title, accepted file types and a submit button.

Introduction

TsgcHTMLComponent_FileUpload emits a complete multipart `<form>` that posts one or more files to the URL given by **Action**. By default it shows a styled drag-and-drop zone (an icon, a title and a subtitle) above a hidden native file input, plus a submit button; set **DragDropEnabled** to False for a plain file input instead. The component builds the markup and its companion CSS for you and returns them through the inherited **HTML** and **CSS** properties.

The drop-zone styling uses Bootstrap variables, so it adapts to light and dark themes. The icon, its size and color, the border style and the button style are all configurable.

Quick Start

1. Drop a **TsgcHTMLComponent_FileUpload** component on the form (or create it at runtime).

2. Set the **Action** URL, optionally **Accept** and **Multiple**, and customise **Title** / **ButtonText**.

3. Read the **HTML** property (and **CSS**) and serve them, or assign them to a page template.

```
TsgcHTMLComponent_FileUpload *oUpload = new TsgcHTMLComponent_FileUpload(NULL);
try
{
  oUpload->Action = "/api/upload";
  oUpload->Accept = "image/*";
  oUpload->Multiple = true;
  oUpload->Title = "Drop images here";
  oUpload->ButtonText = "Upload";
  html = oUpload->HTML;
}
__finally
{
  delete oUpload;
}
```

Form and Files

What it does: *Action* is the URL the form posts to and *InputName* is the field name the file is sent under. *Accept* filters the file picker by type, *Multiple* allows selecting several files and *MaxSize* displays a maximum-size hint. *UploadID* sets the form / input id.

Property	Type	Default	Description
Action	string	'/upload'	URL the multipart form posts the file(s) to.
InputName	string	'file'	The name attribute the file input is submitted under.

Accept	string	''	Accepted file types (the input `accept` attribute, e.g. `image/*`).
Multiple	Boolean	False	When True allows selecting and uploading several files.
MaxSize	string	''	Optional maximum-size text shown as a hint in the drop zone.
UploadID	string	'sgcFileU-pload'	The id of the generated form (and the base of the input id).

Drop Zone and Text

What it does: *DragDropEnabled* shows the styled drop zone (versus a plain file input). *Title* and *Subtitle* are the two lines of text inside the zone, *ButtonText* labels the submit button, and *BorderStyle* sets the CSS border style of the zone.

Property	Type	Default	Description
DragDropEn-abled	Boolean	True	Shows the styled drag-and-drop zone instead of a plain file input.
Title	string	locale 'drop files' text	Main line of text inside the drop zone. Plain text: markup in it is shown literally.
Subtitle	string	locale 'browse' text	Secondary line of text inside the drop zone. Plain text.
ButtonText	string	locale 'upload' text	Caption of the submit button.
BorderStyle	string	'dashed'	CSS border style of the drop zone (for example dashed or solid).

Icon and Button Appearance

What it does: *IconHTML*, *IconSize*, *IconColor* and *IconColorEnum* control the icon in the drop zone. *ButtonStyle* sets the Bootstrap variant of the submit button, and *CSSClass* lets you add extra classes.

Property	Type	Default	Description
IconHTML	string	'📁'	HTML (often an emoji or entity) used as the drop-zone icon.
IconSize	string	'2rem'	CSS font-size of the icon.
IconColor	string	''	Explicit CSS color for the icon. Empty uses IconCol-orEnum.

IconColorEnum	TsgcHTMLColor	hcSecondary	Bootstrap color used for the icon when IconColor is empty.
ButtonStyle	TsgcHTMLButtonStyle	bsPrimary	Bootstrap variant of the submit button.
CSSClass	string	''	Extra CSS classes you can add to the component output.

Output

The inherited **HTML** property returns the complete multipart `<form>` (drop zone, file input and submit button), and the inherited **CSS** property returns the matching `.sgc-upload-zone` rules. Serve both, or assign HTML to the BodyContent and CSS to the styles of a **TsgcHTMLTemplate_Bootstrap**.

Reference

- Properties — 17 properties available on the component.
- Methods — this component adds no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_RichEditor

TsgcHTMLComponent_RichEditor renders a Quill rich-text editor with a configurable toolbar and theme, optional starting content, read-only mode and automatic form submission through a hidden field.

Introduction

TsgcHTMLComponent_RichEditor emits the markup and script that initialise a Quill editor in the browser. You choose a **Toolbar** preset (minimal, basic or full) and a **Theme** (Snow or Bubble), optionally seed it with **Content**, and the component builds the editor, the matching CSS and the Quill initialization for you, returned through the inherited **HTML** and **CSS** properties.

The Quill library and its theme stylesheet are referenced from a CDN. When you set **Name**, a hidden input is added and kept in sync with the editor's HTML on every change, so the formatted content is posted with the surrounding form.

Quick Start

1. Drop a **TsgcHTMLComponent_RichEditor** component on the form (or create it at runtime).

2. Choose a **Toolbar** and **Theme**, set a **Name** for form submission and optionally seed **Content**.

3. Read the **HTML** property (and **CSS**) and serve them, or assign them to a page template.

```
TsgcHTMLComponent_RichEditor *oEditor = new TsgcHTMLComponent_RichEditor(NULL);
try
{
  oEditor->Name = "body";
  oEditor->Toolbar = rtFull;
  oEditor->Theme = reSnow;
  oEditor->Height = "300px";
  oEditor->Content = "<p>Hello</p>";
  html = oEditor->HTML;
}
__finally
{
  delete oEditor;
}
```

Toolbar and Theme

What it does: *Toolbar* selects how many formatting controls the editor shows, and *Theme* selects Quill's Snow (toolbar bar) or Bubble (popover) look. *ReadOnly* renders a non-editable, display-only editor.

Property	Type	Default	Description
Toolbar	TsgcHTMLRichEditorToolbar	rtBasic	Toolbar preset: rtMinimal, rtBasic or rtFull.

Theme	TsgcHTMLRichEditorTheme	reSnow	Quill theme: reSnow (toolbar) or reBubble (popover).
ReadOnly	Boolean	False	When True the editor is display-only and cannot be edited.

Content and Size

What it does: *Content* is the initial HTML shown in the editor, *Placeholder* is the hint shown when it is empty, and *Height* sets the editor's CSS height.

Property	Type	Default	Description
Content	string	''	Initial HTML content placed inside the editor.
Placeholder	string	'Enter text...'	Hint text shown while the editor is empty.
Height	string	'200px'	CSS height of the editor area.

Identity and Form Submission

What it does: *EditorID* sets the id of the editor element (and the prefix of its Quill instance), and *Name* turns on a hidden input that is kept in sync with the editor's HTML so the content is submitted with the form.

Property	Type	Default	Description
EditorID	string	'sgcRichEditor'	The id of the editor element. Use a unique id per editor on a page.
Name	string	''	When set, adds a hidden input of this name carrying the editor HTML for form submission.

```
oEditor->EditorID = "commentEditor";
oEditor->Name = "comment";
```

Output

The inherited **HTML** property returns the Quill stylesheet link, the optional hidden input, the editor `<div>` and the inline initialization `<script>`. The inherited **CSS** property returns the theme-aware overrides for the Quill toolbar and

editor. Serve both, or assign HTML to the BodyContent and CSS to the styles of a **TsgcHTMLTemplate_Bootstrap**.

Reference

- Properties — 8 properties available on the component.
- Methods — this component adds no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_Rating

TsgcHTMLComponent_Rating renders a star (or custom-symbol) rating, with a configurable value, maximum, colors, size and an optional hidden field and value caption.

Introduction

TsgcHTMLComponent_Rating emits a row of symbols, the first **Value** of them drawn filled and the rest empty, out of **MaxValue** in total. The filled and empty symbols, their colors and the symbol size are all configurable, and the component builds the markup for you and returns it through the inherited **HTML** property.

By default the rating is read-only (a display widget). Set **InputName** to add a hidden input carrying the numeric value for form submission, and **ShowValue** to append a small “value / max” caption.

Quick Start

1. Drop a **TsgcHTMLComponent_Rating** component on the form (or create it at runtime).

2. Set the **Value** and **MaxValue**, and optionally the **Color** and **Size**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Rating *oRating = new TsgcHTMLComponent_Rating(NULL);
try
{
  oRating->Value = 4;
  oRating->MaxValue = 5;
  oRating->Color = "#ffc107";
  oRating->ShowValue = true;
  html = oRating->HTML;
}
__finally
{
  delete oRating;
}
```

Value and Behavior

What it does: *Value* is how many symbols are filled and *MaxValue* is the total drawn. *ReadOnly* marks it as a display widget, *InputName* adds a hidden field for form submission, and *ShowValue* appends a numeric caption.

Property	Type	Default	Description
Value	Integer	0	Number of filled symbols (the current rating).
MaxValue	Integer	5	Total number of symbols drawn.
ReadOnly	Boolean	True	Marks the rating as a display-only widget.

Input-Name	string	''	When set, adds a hidden input of this name carrying the value for form submission.
ShowVal-ue	Boolean	False	Appends a small "value / max" caption after the symbols.
RatingID	string	''	Optional id set on the rating container.

Symbols and Appearance

What it does: *FilledSymbol* and *EmptySymbol* are the glyphs used for filled and empty positions, *Size* sets their font size, and the colors come from *Color* / *EmptyColor* (explicit CSS) or, when those are empty, from *ColorStyle* / *EmptyColorStyle* (the Bootstrap palette).

Property	Type	Default	Description
FilledSymbol	string	'★'	Glyph used for a filled position (default a star).
EmptySymbol	string	'★'	Glyph used for an empty position (default a star).
Size	string	'1.5rem'	CSS font-size of the symbols.
Color	string	''	Explicit CSS color for filled symbols. Empty uses ColorStyle.
ColorStyle	Ts-gcHTML-Color	hcWarn-ing	Bootstrap color for filled symbols when Color is empty.
EmptyColor	string	''	Explicit CSS color for empty symbols. Empty uses EmptyCol-orStyle.
EmptyCol-orStyle	Ts-gcHTML-Color	hcLight	Bootstrap color for empty symbols when EmptyColor is empty.

One-Line Build

What it does: The static **Build** class function returns the rating markup in a single call. It is overloaded so the fill color can be passed either as a CSS string or as a **TsgcHTMLColor**.

```
html = TsgcHTMLComponent_Rating::Build(4, 5, "#ffc107");
// or with a Bootstrap color
html = TsgcHTMLComponent_Rating::Build(4, 5, hcWarning);
```

Output

The inherited **HTML** property returns an inline-flex `<div>` containing the optional hidden input, one `<span>` per symbol and the optional value caption. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 13 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_MultiSelect

TsgcHTMLComponent_MultiSelect renders a Bootstrap dropdown multi-select: the chosen options appear as removable chips inside the control, the dropdown can include a search box, and the selection is posted with the form through hidden inputs.

Introduction

TsgcHTMLComponent_MultiSelect emits a `<div class="sgc-multiselect">` containing the control itself, one chip per selected option, the dropdown with the option buttons, a “no results” row, the hidden inputs and a small inline script that drives the whole widget in the browser. No external JavaScript library is required.

Options live in the **Options** collection, each one a **TsgcHTMLMultiSelectOption** with *Value* (what is posted), *Text* (what the user reads), *Selected*, *Disabled* and *Group*. Fill the collection at design time, add options one by one with **AddOption**, or load them straight from a dataset with **LoadFromDataSet**. The complete markup is returned through the inherited **HTML** property.

Quick Start

1. Drop a **TsgcHTMLComponent_MultiSelect** component on the form (or create it at runtime).

2. Set the **FieldName** so the selection is posted, then add the options with **AddOption**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_MultiSelect *oMultiSelect = new TsgcHTMLComponent_MultiSelect(NULL);
try
{
  oMultiSelect->FieldName = "skills";
  oMultiSelect->MaxSelections = 3;
  oMultiSelect->AddOption("delphi", "Delphi", true);
  oMultiSelect->AddOption("cpp", "C++ Builder");
  oMultiSelect->AddOption("csharp", "C#");
  html = oMultiSelect->HTML;
}
__finally
{
  delete oMultiSelect;
}
```

Options

What it does: *Options* is the collection of selectable entries. Every item is a **TsgcHTMLMultiSelectOption** with *Value* (posted with the form), *Text* (shown in the dropdown and in the chip), *Selected* (starts selected, so a chip is rendered for it), *Disabled* (rendered greyed out and cannot be toggled) and *Group*. A non-empty *Group* starts a new `dropdown-header` heading in the dropdown, so add the options already ordered by group. Use **AddOption** for the common case of value plus text.

Property	Type	Default	Description

Options	TsgcHTMLMultiSelectOptions	(empty)	Collection of TsgcHTMLMultiSelectOption items shown in the dropdown.

Texts

What it does: *Placeholder* is the muted text shown in the control while nothing is selected, *SearchPlaceholder* is the placeholder of the search box inside the dropdown, and *NoResultsText* is the row shown when the search box matches no option. All three default to an empty string and are resolved **at render time** from the active locale (`sgcHTMLDefaultLocale`, unit `sgcHTML_Locale`, with a per-request override through `sgcHTMLSetRequestLocale`). The English locale supplies *Select...*, *Search* and *No results*. Set any of them to a non-empty value and that value always wins, in every language.

Property	Type	Default	Description
Placeholder	string	(locale)	Text shown when nothing is selected. English locale default 'Select...'.
SearchPlaceholder	string	(locale)	Placeholder of the search box. English locale default 'Search'. Only rendered when ShowSearch is True.
NoResultsText	string	(locale)	Text shown when the search box matches nothing. English locale default 'No results'.

The search box placeholder used to be a hard-coded English `Search...` that bypassed the locale. It now resolves like every other caption here: *SearchPlaceholder* when you set one, otherwise the active locale's `SearchText`.

Search, Limits and Size

What it does: *ShowSearch* adds a filter box at the top of the dropdown that hides the options which do not match what the user types (group headings are hidden while a filter is active, and the *NoResultsText* row appears when nothing matches). *MaxSelections* caps how many options can be selected at the same time; once the cap is reached the script simply refuses further selections. *Size* maps to the Bootstrap select sizes.

Property	Type	Default	Description
ShowSearch	Boolean	True	Shows the search / filter box inside the dropdown.
SearchPlaceholder	string	(locale)	Placeholder of that search box. Empty means the active locale's SearchText.
MaxSelections	Integer	0	Maximum number of options that can be selected. 0 means unlimited.
Size	TsgcHTMLMultiSelectSize	mssDefault	Control size: mssDefault, mssSmall (form-select-sm) or mssLarge (form-select-lg).

Form Integration

What it does: When *FieldName* is set, one hidden `<input>` per selected option is emitted with that name, so a normal form POST delivers the selection as a repeated field. The script keeps the hidden inputs in sync with every click. When *FieldName* is empty, no hidden inputs are rendered and the control is display only.

MultiSelectID is the id of the root element, and every inner element derives its id from it. It is resolved at render time: the property when you set it, otherwise `msel_` plus the *FieldName*, and when both are empty a process-unique id that is generated on the first render and cached for the life of the instance. Leaving it empty is therefore safe, two multi-selects on the same page never collide, and the id also stays stable across re-renders and HTMX out-of-band swaps. Set it explicitly when you need a known id, for example to reach the element from your own JavaScript.

Property	Type	Default	Description
FieldName	string	''	Name of the hidden inputs posted with the form, one per selected option. Empty means no hidden inputs.
MultiSelectID	string	(empty)	id of the root element. Empty means 'msel_' + FieldName, or a unique auto id when FieldName is empty too.

Data Binding

What it does: The component is data aware. **LoadFromDataSet** clears the *Options* collection and walks the dataset, adding one option per record from a value field, a text field and an optional selected field. It also remembers those field names, so the inherited *DataSource* plus *DataAutoRefresh* path (`DoLoadFromDataSource`) re-runs **LoadFromDataSet** with the same fields whenever the data changes.

```
oMultiSelect->LoadFromDataSet(qrySkills, "code", "name", "assigned");
```

Output

The inherited **HTML** property returns the `<div class="sgc-multiselect">` root with the control, the chips, the dropdown (search box, options, group headings and the no-results row), the hidden inputs and the inline script. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 9 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Transfer

TsgcHTMLComponent_Transfer renders a dual list (a “pick list”): an available panel on the left, a selected panel on the right and a column of buttons in between to move items from one side to the other. The items that end up on the right are posted with the form through hidden inputs.

Introduction

TsgcHTMLComponent_Transfer emits a `<div class="sgc-transfer">` holding two Bootstrap cards, each with a heading, an optional filter box and a scrollable list, the four move buttons between them, the hidden inputs and a small inline script that drives the widget in the browser. No external JavaScript library is required.

Items live in the **Items** collection, each one a **TsgcHTMLTransferItem** with *Value* (what is posted), *Text* (what the user reads), *Assigned* (when True the item starts in the right panel) and *Disabled* (the item is rendered greyed out and can never be moved). Fill the collection at design time, add items one by one with **AddItem**, or load them straight from a dataset with **LoadFromDataSet**. The complete markup is returned through the inherited **HTML** property.

In the browser a single click selects an item, a double click moves it to the other panel, and the four buttons move either the selected items or every item of a panel.

Quick Start

1. Drop a **TsgcHTMLComponent_Transfer** component on the form (or create it at runtime).

2. Set the **FieldName** so the assigned items are posted, then add the items with **AddItem**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Transfer *oTransfer = new TsgcHTMLComponent_Transfer(NULL);
try
{
  oTransfer->FieldName = "roles";
  oTransfer->ShowFilter = true;
  oTransfer->AddItem("admin", "Administrator");
  oTransfer->AddItem("editor", "Editor", true);
  oTransfer->AddItem("viewer", "Viewer");
  html = oTransfer->HTML;
}
__finally
{
  delete oTransfer;
}
```

Items

What it does: *Items* is the collection of transferable entries. Every item is a **TsgcHTMLTransferItem** with *Value* (posted with the form), *Text* (shown in the list), *Assigned* (when True the item is rendered in the right panel instead of the left one) and *Disabled* (rendered greyed out; the script refuses to select or move it, and even “move all” skips it). Use **AddItem** for the common case of value plus text.

Property	Type	Default	Description
Items	TsgcHTMLTrans-ferItems	(empty)	Collection of TsgcHTMLTransferItem entries shown in the two panels.

Panel Titles

What it does: *TitleSource* is the heading of the left (available) panel and *TitleTarget* the heading of the right (selected) panel. Both default to an empty string and are resolved **at render time** from the active locale (`sgcHTMLDefaultLocale`, unit `sgcHTML_Locale`, with a per-request override through `sgcHTMLSetRequestLocale`). The English locale supplies *Available* and *Selected*. Set either property to a non-empty value and that value always wins, in every language.

Property	Type	Default	Description
TitleSource	string	(locale)	Heading of the left / available panel. English locale default 'Available'.
TitleTarget	string	(locale)	Heading of the right / selected panel. English locale default 'Selected'.

Buttons

What it does: Four buttons sit between the panels: move the selected items to the right, move the selected items back to the left, move every item to the right and move every item back to the left. Unlike the panel titles, these captions are **not** locale driven: the constructor gives them the literal arrow-like defaults listed below, and you change them by assigning your own text (a word, an arrow character, or an icon markup fragment is up to you).

Property	Type	Default	Description
AddCaption	string	'>'	Caption of the button that moves the selected items to the right panel.
RemoveCaption	string	'<'	Caption of the button that moves the selected items back to the left panel.
AddAllCaption	string	'>>'	Caption of the button that moves every item to the right panel.
RemoveAllCaption	string	'<<'	Caption of the button that moves every item back to the left panel.

The two "move all" buttons never move a *Disabled* item, and they skip the items that the filter is currently hiding, so filtering first and then pressing "move all" moves exactly the visible items.

Filter and Layout

What it does: *ShowFilter* adds a small text box above each panel that hides the items which do not match what the user types (the match is case insensitive and applies to that panel only). It is **False** by default, so no filter box is rendered until you switch it on. *FilterPlaceholder* is the placeholder of those filter boxes. *Height* is the CSS height of each list; the panel scrolls when the items do not fit.

Property	Type	Default	Description
ShowFilter	Boolean	False	Adds a filter text box above each panel.
FilterPlaceholder	string	(locale)	Placeholder of both filter boxes. English locale default 'Filter...'. Only rendered when ShowFilter is True.
Height	string	'220px'	CSS height of each list panel. The panel scrolls vertically when the items overflow.

The filter placeholder used to be a hard-coded English `Filter...` that bypassed the locale. Like the panel titles, it now resolves **at render time**: *FilterPlaceholder* when you set one, otherwise the active locale's `FilterText`.

Form Integration

What it does: When *FieldName* is set, one hidden `<input>` per assigned item (that is, per item currently in the right panel) is emitted with that name, so a normal form POST delivers the selection as a repeated field. The script rebuilds those inputs after every move. When *FieldName* is empty, no hidden inputs are rendered and the control is display only.

TransferID is the id of the root element, and every inner element derives its id from it. It is resolved at render time: the property when you set it, otherwise `tr_` plus the *FieldName*, and when both are empty a process-unique id that is generated on the first render and cached for the life of the instance. Leaving it empty is therefore safe, two transfers on the same page never collide, and the id also stays stable across re-renders and HTMX out-of-band swaps. Set it explicitly when you need a known id, for example to reach the element from your own JavaScript.

Property	Type	Default	Description
FieldName	string	''	Name of the hidden inputs posted for each assigned item. Empty means no hidden inputs.
TransferID	string	(empty)	id of the root element. Empty means 'tr_' + FieldName, or a unique auto id when FieldName is empty too.

Data Binding

What it does: The component is data aware. **LoadFromDataSet** clears the *Items* collection and walks the dataset, adding one item per record from a value field, a text field and an optional assigned field. It also remembers those

field names, so the inherited *DataSource* plus *DataAutoRefresh* path (`DoLoadFromDataSource`) re-runs **LoadFromDataSet** with the same fields whenever the data changes.

```
oTransfer->LoadFromDataSet(qryRoles, "code", "name", "granted");
```

Output

The inherited **HTML** property returns the `<div class="sgc-transfer">` root with the two panels (heading, optional filter box and scrollable `<ul>` list), the column with the four move buttons, the hidden inputs and the inline script. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 12 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_TimePicker

TsgcHTMLComponent_TimePicker renders a labeled Bootstrap time input (`<input type="time">`) with optional minimum, maximum and step, ready to be posted with a form.

Introduction

TsgcHTMLComponent_TimePicker emits a single HTML time field: an optional `<label>` followed by an `<input type="time">` element, wrapped in a Bootstrap `mb-3` column. The browser supplies the clock user interface, so no JavaScript library and no inline script are required.

The markup is built through the sgcHTML node layer (**TsgcHTMLField**), which emits the label and the input and encodes every value exactly once, so a time, a label or a field name coming from user data cannot break out of the attribute it is written into.

Quick Start

1. Drop a **TsgcHTMLComponent_TimePicker** component on the form (or create it at runtime).

2. Set **FieldName** (the name posted with the form), **LabelText** and optionally an initial **Value**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_TimePicker *oPicker = new TsgcHTMLComponent_TimePicker(NULL);
try
{
  oPicker->FieldName = "start_time";
  oPicker->LabelText = "Start Time";
  oPicker->Value = "09:00";
  oPicker->MinTime = "08:00";
  oPicker->MaxTime = "20:00";
  html = oPicker->HTML;
}
__finally
{
  delete oPicker;
}
```

Value and Range

What it does: *Value* is the time the field shows when the page loads. *MinTime* and *MaxTime* restrict the times the browser accepts, and are written as the `min` and `max` attributes of the input. All three use the HTML time format, `HH:MM` or `HH:MM:SS` in 24 hour notation, whatever the clock the browser displays to the user.

Property	Type	Default	Description
Value	string	''	Initial time shown by the field, `HH:MM` or `HH:MM:SS`. Empty leaves the field blank.

MinTime	string	''	Earliest selectable time. Empty omits the `min` attribute.
MaxTime	string	''	Latest selectable time. Empty omits the `max` attribute.

Seconds and Step

What it does: By default a browser time input works in whole minutes. *StepSeconds* writes an explicit `step` attribute, in seconds, so the field snaps to your own increment (for example 900 for quarters of an hour). *ShowSeconds* is the shortcut for the common case: when it is True and *StepSeconds* is 0 or less, the component emits `step="1"` so the browser also shows a seconds box. An explicit *StepSeconds* always wins over *ShowSeconds*.

Property	Type	Default	Description
StepSeconds	Integer	0	Value of the `step` attribute, in seconds. 0 leaves the browser default of one minute.
ShowSeconds	Boolean	False	When True and StepSeconds is 0 or less, emits `step="1"` so the browser shows seconds.

Form Integration

What it does: *FieldName* is the `name` attribute of the input, which is the key the value is posted under, and it also drives the element id, which is `'tp_'` followed by the field name. *LabelText* renders a `form-label` above the input and is tied to it through the `for` attribute. *Required* and *Disabled* add the matching HTML attributes.

Property	Type	Default	Description
FieldName	string	''	The `name` attribute of the input. The id is derived from it as `'tp_' + FieldName`.
LabelText	string	''	Label rendered above the input. Empty means no label element at all.
Required	Boolean	False	Adds the `required` attribute, so the browser blocks an empty submit.
Disabled	Boolean	False	Adds the `disabled` attribute. A disabled input is not posted.

Size

What it does: *Size* picks the Bootstrap input size. The enumeration has five values, while Bootstrap only defines three input sizes, so the values collapse: `hsExtraSmall` and `hsSmall` both render `form-control-sm`, `hsMedium` renders the plain `form-control`, and `hsLarge` and `hsExtraLarge` both render `form-control-lg`.

Property	Type	Default	Description
Size	TsgcHTMLSize	hsMedium	Bootstrap input size: hsExtraSmall, hsSmall, hsMedium, hsLarge or hsExtraLarge.

Markup in One Call

What it does: When a page just needs a plain time field, the class function **Build** saves the create / configure / free dance. It creates a temporary picker, returns its markup and frees it, all in a single expression.

```
UnicodeString html =
  TsgcHTMLComponent_TimePicker::Build("start_time", "Start Time", "09:00");
```

Output

The inherited **HTML** property returns a `<div class="mb-3">` containing the optional `<label class="form-label">` and the `<input type="time">` element, with the `min`, `max` and `step` attributes filled in from the properties above. Serve this string directly, place it inside a form, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

```
<div class="mb-3">
  <label class="form-label" for="tp_start_time">Start Time</label>
  <input type="time" id="tp_start_time" name="start_time" class="form-control"
         value="09:00" min="08:00" max="20:00">
</div>
```

Reference

- Properties — 10 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_DateTimePicker

TsgcHTMLComponent_DateTimePicker renders a labeled Bootstrap date and time input (`<input type="datetime-local">`) with an optional minimum and maximum, ready to be posted with a form.

Introduction

TsgcHTMLComponent_DateTimePicker emits a single HTML field that captures a date and a time together: an optional `<label>` followed by an `<input type="datetime-local">` element, wrapped in a Bootstrap `mb-3` column. The browser supplies the calendar and clock user interface, so no JavaScript library and no inline script are required.

The markup is built through the sgcHTML node layer (**TsgcHTMLField**), which emits the label and the input and encodes every value exactly once, so a date, a label or a field name coming from user data cannot break out of the attribute it is written into.

Use this component when a single instant matters (an appointment, a deadline, a publish-at moment). When only the clock time matters, use TsgcHTMLComponent_TimePicker instead.

Quick Start

1. Drop a **TsgcHTMLComponent_DateTimePicker** component on the form (or create it at runtime).

2. Set **FieldName** (the name posted with the form), **LabelText** and optionally an initial **Value**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_DateTimePicker *oPicker =
  new TsgcHTMLComponent_DateTimePicker(NULL);
try
{
  oPicker->FieldName = "appointment";
  oPicker->LabelText = "Appointment";
  oPicker->Value = "2026-07-15T09:30";
  oPicker->MinValue = "2026-07-01T08:00";
  oPicker->MaxValue = "2026-07-31T20:00";
  html = oPicker->HTML;
}
__finally
{
  delete oPicker;
}
```

Value and Range

What it does: *Value* is the date and time the field shows when the page loads. *MinValue* and *MaxValue* restrict what the browser accepts, and are written as the `min` and `max` attributes of the input. All three use the HTML local date and time format, `YYYY-MM-DDTHH:MM`, with a literal `T` between the date and the time and the time in 24 hour notation, whatever the calendar the browser displays to the user.

Property	Type	Default	Description

Value	string	''	Initial date and time shown by the field, `YYYY-MM-DDTHH:MM`. Empty leaves the field blank.
MinValue	string	''	Earliest selectable date and time. Empty omits the `min` attribute.
MaxValue	string	''	Latest selectable date and time. Empty omits the `max` attribute.

Form Integration

What it does: *FieldName* is the `name` attribute of the input, which is the key the value is posted under, and it also drives the element id, which is `'dtp_'` followed by the field name. *LabelText* renders a `form-label` above the input and is tied to it through the `for` attribute. *Required* and *Disabled* add the matching HTML attributes.

Property	Type	Default	Description
FieldName	string	''	The `name` attribute of the input. The id is derived from it as `'dtp_' + FieldName`.
LabelText	string	''	Label rendered above the input. Empty means no label element at all.
Required	Boolean	False	Adds the `required` attribute, so the browser blocks an empty submit.
Disabled	Boolean	False	Adds the `disabled` attribute. A disabled input is not posted.

Size

What it does: *Size* picks the Bootstrap input size. The enumeration has five values, while Bootstrap only defines three input sizes, so the values collapse: `hsExtraSmall` and `hsSmall` both render `form-control-sm`, `hsMedium` renders the plain `form-control`, and `hsLarge` and `hsExtraLarge` both render `form-control-lg`.

Property	Type	Default	Description
Size	TsgcHTMLSize	hsMedium	Bootstrap input size: hsExtraSmall, hsSmall, hsMedium, hsLarge or hsExtraLarge.

Markup in One Call

What it does: When a page just needs a plain date and time field, the class function **Build** saves the create / configure / free dance. It creates a temporary picker, returns its markup and frees it, all in a single expression.

```
UnicodeString html = TsgcHTMLComponent_DateTimePicker::Build("appointment",
  "Appointment", "2026-07-15T09:30");
```

Output

The inherited **HTML** property returns a `<div class="mb-3">` containing the optional `<label class="form-label">` and the `<input type="datetime-local">` element, with the `min` and `max` attributes filled in from the properties above. Serve this string directly, place it inside a form, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

```
<div class="mb-3">
  <label class="form-label" for="dtp_appointment">Appointment</label>
  <input type="datetime-local" id="dtp_appointment" name="appointment"
         class="form-control" value="2026-07-15T09:30"
         min="2026-07-01T08:00" max="2026-07-31T20:00">
</div>
```

The posted value is a *local* date and time. It carries no time zone and no offset, so the server decides which zone it belongs to, usually the zone of the user or of the tenant.

Reference

- Properties — 8 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_DateRangePicker

TsgcHTMLComponent_DateRangePicker renders two linked date inputs (a start date and an end date) with optional Today / Last 7 days / Last 30 days / This month preset buttons.

Introduction

TsgcHTMLComponent_DateRangePicker emits two labelled `<input type="date">` fields wrapped in a single container, plus a small inline script that keeps the range consistent while the user picks dates. The two inputs are ordinary form fields, so the selected range is posted with the form under the names given by **FieldNameStart** and **FieldNameEnd**. The complete markup is returned by the inherited **HTML** property.

The inline script links the two inputs: choosing a start date raises the end input's `min` attribute, and choosing an end date lowers the start input's `max` attribute, so the browser itself prevents an inverted range. Clearing an input restores the original bound. **MinDate** and **MaxDate** are global bounds applied to both inputs from the server.

The labels and the preset captions are left empty by the constructor and are resolved from the active locale at render time, so the same component instance can serve concurrent requests in different languages.

Quick Start

1. Drop a **TsgcHTMLComponent_DateRangePicker** component on the form (or create it at runtime).

2. Set the field names, the initial values and (optionally) **ShowPresets**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_DateRangePicker *oRange = new TsgcHTMLComponent_DateRangePicker(NULL);
try
{
  oRange->FieldNameStart = "from";
  oRange->FieldNameEnd = "to";
  oRange->StartValue = "2026-01-01";
  oRange->EndValue = "2026-01-31";
  oRange->MinDate = "2020-01-01";
  oRange->MaxDate = "2030-12-31";
  oRange->ShowPresets = true;
  html = oRange->HTML;
}
__finally
{
  delete oRange;
}
```

Dates and Bounds

What it does: *StartValue* and *EndValue* pre-fill the two inputs, and *MinDate* and *MaxDate* restrict what the user can pick. All four use the HTML date format `YYYY-MM-DD`. MinDate and MaxDate are written to both inputs, and the inline script then tightens them further as the user picks a start or an end date.

Property	Type	Default	Description
StartValue	string	''	Initial start date (YYYY-MM-DD). Empty leaves the start input blank.
EndValue	string	''	Initial end date (YYYY-MM-DD). Empty leaves the end input blank.
MinDate	string	''	Global lower bound applied to both inputs.
MaxDate	string	''	Global upper bound applied to both inputs.

Form Fields and Layout

What it does: *FieldNameStart* and *FieldNameEnd* are the `name` attributes of the two inputs, so the picked range arrives at the server as two ordinary form values. The element ids are derived from them: `drp_` + FieldNameStart and `drp_` + FieldNameEnd. *Layout* selects whether the two fields sit side by side (`drlInline`) or one under the other (`drlStacked`).

Property	Type	Default	Description
FieldNameStart	string	'range_start'	name attribute of the start input; its id is 'drp_' + FieldNameStart.
FieldNameEnd	string	'range_end'	name attribute of the end input; its id is 'drp_' + FieldNameEnd.
Layout	TsgcHTMLDateRangeLayout	drlInline	drlInline puts the two inputs side by side in a row; drlStacked stacks them.

Captions and Presets

What it does: *LabelStart* and *LabelEnd* are the labels drawn above the two inputs. *ShowPresets* adds a small button group under the fields with four shortcuts that fill both inputs client-side. The four preset captions only render when ShowPresets is True. All six captions default to an empty string and fall back to the active locale (`sgcHTMLDefaultLocale`, unit `sgcHTML_Locale`) at render time; a value you set always wins.

Property	Type	Default	Description
LabelStart	string	(locale)	Label of the start input. English locale default: From.
LabelEnd	string	(locale)	Label of the end input. English locale default: To.
ShowPresets	Boolean	False	Shows the Today / Last 7 days / Last 30 days / This month button group.

PresetTodayCaption	string	(locale)	Caption of the first preset button. English locale default: Today.
PresetLast7Caption	string	(locale)	Caption of the second preset button. English locale default: Last 7 days.
PresetLast30Caption	string	(locale)	Caption of the third preset button. English locale default: Last 30 days.
PresetThisMonth-Caption	string	(locale)	Caption of the fourth preset button. English locale default: This month.

```
// Stacked layout, presets shown, captions taken from the active locale
oRange->Layout = drlStacked;
oRange->ShowPresets = true;
```

Output

The inherited **HTML** property returns a `<div>` containing the two date fields (inside a `row` when Layout is drlInline), the optional preset button group and the inline script that links the two inputs. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

This component is a plain form control: it is not data-aware and has no LoadFromDataSet method.

Reference

- Properties — 14 properties available on the component.
- Methods — this component publishes no methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_Slider

TsgcHTMLComponent_Slider renders a Bootstrap range input with an optional label, an optional live value badge and optional min/max end captions.

Introduction

TsgcHTMLComponent_Slider emits an HTML `<input type="range">` element styled with the Bootstrap `form-range` class, wrapped in a `<div>` that can also carry a label, a colored badge showing the current value and a pair of small captions with the lowest and highest selectable values. You define the range with **Min**, **Max** and **Step**, set the initial position with **Value**, and read the finished markup from the inherited **HTML** property.

The component is a server-side HTML generator. It does not require any JavaScript library: when **ShowValue** is enabled it emits a small inline `oninput` handler that copies the current value into the badge while the user drags the handle, and nothing else.

Quick Start

1. Drop a **TsgcHTMLComponent_Slider** component on the form (or create it at runtime).

2. Set the range with **Min**, **Max** and **Step**, the starting position with **Value**, and a **FieldName** so the value is posted with the form.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Slider *oSlider = new TsgcHTMLComponent_Slider(NULL);
try
{
  oSlider->SliderID = "volume";
  oSlider->LabelText = "Volume";
  oSlider->FieldName = "volume";
  oSlider->Min = 0;
  oSlider->Max = 100;
  oSlider->Step = 5;
  oSlider->Value = 40;
  oSlider->ShowValue = true;
  oSlider->ShowMinMax = true;
  oSlider->Color = "#7C3AED";
  html = oSlider->HTML;
}
__finally
{
  delete oSlider;
}
```

Range and Value

What it does: *Min* and *Max* define the two ends of the track, *Step* is the increment the handle snaps to, and *Value* is the position the slider starts at. The three of them are written straight into the `min`, `max`, `step` and `value` attributes of the range input, so the browser enforces them.

Property	Type	Default	Description
Min	Integer	0	Lowest selectable value.
Max	Integer	100	Highest selectable value.
Step	Integer	1	Increment between values.
Value	Integer	0	Initial value of the slider.

Label and Readouts

What it does: *LabelText* renders a `<label>` to the left of the track, bound to the range input. *ShowValue* renders a colored badge to the right of the label that a small inline `oninput` handler updates live while the user drags the handle. *ShowMinMax* adds two small muted captions under the track, showing the *Min* and *Max* values.

Property	Type	Default	Description
LabelText	string	''	Label rendered to the left of the badge. Empty means no label.
ShowValue	Boolean	True	Shows a colored badge with the live value, updated as the user drags.
ShowMinMax	Boolean	False	Shows small muted Min and Max captions under the track.

Color

What it does: The slider has a single accent color, applied as the CSS `accent-color` of the range input and as the background of the live value badge. *Color* is a raw CSS color string and *ColorStyle* is a Bootstrap contextual color. When *Color* is not empty it always wins, and *ColorStyle* is ignored.

Property	Type	Default	Description
Color	string	''	Explicit CSS accent color, for example '#7C3AED'. When set it overrides ColorStyle.
ColorStyle	TsgcHTMLColor	hcPrimary	Bootstrap contextual accent color, used only when Color is empty. hcPrimary is #0d6efd.

Form Integration

What it does: *FieldName* becomes the `name` attribute of the range input, so the selected value is posted together with the rest of the form. *Disabled* adds the `disabled` attribute, which greys the track out and stops the browser from posting the value.

Property	Type	Default	Description
FieldName	string	''	name attribute of the range input, so the value is posted with the form.
Disabled	Boolean	False	Adds the disabled attribute to the range input.

Element Ids

What it does: *SliderID* is the id of the root `<div>`. The range input gets the id `<SliderID>_input` and the value badge gets `<SliderID>_value`, which is what the inline `oninput` handler looks up with `document.getElementById` and updates.

The id is resolved at render time: *SliderID* when you set it, otherwise `sld_` plus the *FieldName*, and when both are empty a process-unique id that is generated on the first render and cached for the life of the instance. Leaving it empty is therefore safe, two sliders on the same page never collide, and the id also stays stable across re-renders and HTMX out-of-band swaps. Set it explicitly when you need a known id, for example to reach the element from your own JavaScript.

Property	Type	Default	Description
SliderID	string	(empty)	id of the root element. Empty means 'sld_' + FieldName, or a unique auto id when FieldName is empty too. The input is <id>_input and the badge <id>_value.

Orientation and Dual Thumb

What it does: *Orientation* lays the track out horizontally or vertically (minimum at the bottom). *DualThumb* turns the slider into a range selector with two thumbs, *ValueLow* and *ValueHigh*, and the filled track between them; the two values are posted as `<FieldName>_low` and `<FieldName>_high`. With the defaults the output is the classic single thumb slider.

Property	Type	Default	Description
Orientation	TsgcHTMLSliderOrientation	sloHorizontal	Horizontal or vertical track.
DualThumb	Boolean	False	Two thumbs select a range (ValueLow..ValueHigh) instead of a single Value.

ValueLow	Integer	25	Low value of a DualThumb slider.
ValueHigh	Integer	75	High value of a DualThumb slider.

Live Posting

What it does: Without these properties the value is posted only with the surrounding form. *PostURL* adds `hx-post` with `hx-trigger="change"`, so the server receives the value each time the thumb is released; *PostTarget* chooses where the response goes (by default it is discarded). *WSSend* sends the value over the htmx WebSocket extension instead. For a dual thumb slider every request carries both values.

Property	Type	Default	Description
PostURL	string	'' (empty)	URL that receives an hx-post when the thumb is released.
PostTarget	string	'' (empty)	hx-target for the response of PostURL; empty discards the response.
WSSend	Boolean	False	Sends the change through the htmx WebSocket extension instead of hx-post.

Live Updates

When the value changes on the server, call **GetLiveValueFragment** (or **GetLiveValuesFragment** for a dual thumb slider) and broadcast the result with **TsgcHTMX_Engine_Server.BroadcastFragment**: every browser swaps the slider in place. Set *SliderID* so the page and the updates use the same id. **TsgcHTMLInstrumentsMQTTBinder** can drive a slider from an MQTT topic with the ibtValue target. See Instruments.

Output

The inherited **HTML** property returns a `<div>` containing the optional label and value badge, the `<input type="range">` element and the optional min/max captions. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 19 properties available on the component.
- Methods — 2 public methods, the live update helpers.
- Events — this component has no published events.

TsgcHTMLComponent_RangeSlider

TsgcHTMLComponent_RangeSlider renders a two-handle range control that selects a low and a high bound, with a filled track between the handles, optional value labels and hidden inputs so both bounds post with the form.

Introduction

TsgcHTMLComponent_RangeSlider emits two overlapping `<input type="range">` elements, one for the low handle and one for the high handle, drawn on top of a shared track. The part of the track between the two handles is filled, so the selected interval is visible at a glance. Use it for price ranges, age brackets, score filters and any other value that has two bounds instead of one.

An inline script keeps the two handles in order. It moves the fill between them, refreshes the value labels, writes both bounds into the hidden inputs and, when **MinGap** is set, pushes the other handle away instead of letting the two cross. The complete markup is returned by the inherited **HTML** property.

The component also ships its own CSS, so the track, the fill and the round handles are styled without any extra stylesheet. Unlike TsgcHTMLComponent_Slider, the range slider has no color property: the fill and the handles are drawn in the Bootstrap primary blue, `#0d6efd`.

Quick Start

1. Drop a **TsgcHTMLComponent_RangeSlider** component on the form (or create it at runtime).

2. Set the range with **Min**, **Max** and **Step**, the starting bounds with **ValueLow** and **ValueHigh**, and the two field names so both bounds are posted.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_RangeSlider *oRange = new TsgcHTMLComponent_RangeSlider(NULL);
try
{
  oRange->RangeSliderID = "price";
  oRange->Min = 0;
  oRange->Max = 500;
  oRange->Step = 10;
  oRange->ValueLow = 100;
  oRange->ValueHigh = 350;
  oRange->MinGap = 50;
  oRange->ShowValues = true;
  oRange->FieldNameLow = "price_min";
  oRange->FieldNameHigh = "price_max";
  html = oRange->HTML;
}
__finally
{
  delete oRange;
}
```

Range and Bounds

What it does: *Min* and *Max* define the two ends of the track and are applied to both handles. *Step* is the increment both handles snap to. *ValueLow* and *ValueHigh* are the positions the two handles start at. *MinGap* is the minimum distance the script keeps between them.

Property	Type	Default	Description
Min	Integer	0	Lowest selectable value, applied to both handles.
Max	Integer	100	Highest selectable value, applied to both handles.
Step	Integer	1	Increment between values, applied to both handles.
ValueLow	Integer	25	Initial position of the low handle.
ValueHigh	Integer	75	Initial position of the high handle.
MinGap	Integer	0	Minimum distance kept between the two handles. The script pushes the other handle away rather than letting them cross.

Readouts

What it does: *ShowValues* renders the low and the high value as small muted labels above the track, one on each side. The inline script updates them while the user drags, so they always show the current interval.

Property	Type	Default	Description
ShowValues	Boolean	True	Shows the low and high values as small muted labels above the track.

Form Integration

What it does: The two range inputs carry no `name` attribute. Instead, the bounds are posted through two hidden inputs named after *FieldNameLow* and *FieldNameHigh*, which the inline script keeps in sync with the handles. Leave a field name empty and that bound is simply not posted.

Disabled adds the `disabled` attribute to both handles **and to both hidden inputs**. Since the hidden inputs are the only elements here that carry a `name`, and the browser skips disabled controls when it builds the submission set, a disabled range slider posts **nothing**. Earlier versions left the hidden inputs enabled, so a disabled range slider still posted its interval; that is no longer the case. Render your own hidden inputs next to the component if the server must still receive a fixed interval.

Property	Type	Default	Description
FieldNameLow	string	''	name of the hidden input carrying the low value. Empty means the low bound is not posted.

FieldName-High	string	''	name of the hidden input carrying the high value. Empty means the high bound is not posted.
Disabled	Boolean	False	Disables both handles and both hidden inputs, so nothing is posted.

Element Ids

What it does: *RangeSliderID* is the id of the root `<div>` and every inner element derives its id from it: `_low` and `_high` for the two range inputs, `_fill` for the filled track, `_vlow` and `_vhigh` for the value labels, `_flow` and `_fhigh` for the hidden inputs. The inline script looks all of them up with `document.getElementById`.

The id is resolved at render time: *RangeSliderID* when you set it, otherwise `rsld_` plus *FieldNameLow*, and when both are empty a process-unique id that is generated on the first render and cached for the life of the instance. Leaving it empty is therefore safe, two range sliders on the same page never collide, and the id also stays stable across re-renders and HTMX out-of-band swaps. Set it explicitly when you need a known id, for example to reach the element from your own JavaScript.

Property	Type	Default	Description
RangeSliderID	string	(empty)	id of the root element. Empty means 'rsld_' + FieldNameLow, or a unique auto id when FieldNameLow is empty too.

Output

The inherited **HTML** property returns a `<div class="sgc-range-slider">` containing the optional value labels, the track, the fill, the two overlapping range inputs and the optional hidden inputs, followed by the inline script that wires them together. The component also emits its own CSS, which draws the track, the fill and the round handles in `#0d6efd`. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 11 properties available on the component.
- Methods — this component declares no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_ColorPicker

TsgcHTMLComponent_ColorPicker renders a Bootstrap color input with an optional hex text box kept in sync with it and an optional row of clickable swatch buttons.

Introduction

TsgcHTMLComponent_ColorPicker emits an `<input type="color">` styled as a Bootstrap `form-control-color`, optionally preceded by a label and optionally followed by a small text box showing the hex value. It is an ordinary form field: when **FieldName** is set, the selected color is posted with the surrounding form. The complete markup is returned by the inherited **HTML** property.

Colors are always expressed in the `#rrggbb` form the browser's native color input uses. The default is `#0d6efd`, the Bootstrap primary blue.

Quick Start

1. Drop a **TsgcHTMLComponent_ColorPicker** component on the form (or create it at runtime).

2. Set **FieldName**, **Value** and (optionally) **LabelText**, **ShowValueInput** and a list of **Swatches**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_ColorPicker *oPicker = new TsgcHTMLComponent_ColorPicker(NULL);
try
{
  oPicker->FieldName = "brand_color";
  oPicker->LabelText = "Brand color";
  oPicker->Value = "#7C3AED";
  oPicker->ShowValueInput = true;
  oPicker->Swatches->Add("#0d6efd");
  oPicker->Swatches->Add("#198754");
  oPicker->Swatches->Add("#dc3545");
  html = oPicker->HTML;
}
__finally
{
  delete oPicker;
}
```

Color and Label

What it does: *Value* is the selected color, *LabelText* is the caption drawn above the picker and *FieldName* is the name under which the color is posted. *Disabled* greys out the control.

Property	Type	Default	Description
Value	string	'#0d6efd'	Selected color, in #rrggbb form.
LabelText	string	''	Label rendered above the picker. Empty = no label.

FieldName	string	''	name attribute of the color input. Empty = the color is not posted.
Disabled	Boolean	False	Disables the picker and the hex text box. Swatch clicks do nothing while it is True.
ColorPickerID	string	(empty)	id of the root element. Empty means 'cp_' + FieldName, or a unique auto id when FieldName is empty too.

The root id is resolved at render time: *ColorPickerID* when you set it, otherwise `cp_` plus the *FieldName*, and when both are empty a process-unique id that is generated on the first render and cached for the life of the instance. Leaving it empty is therefore safe, two pickers on the same page never collide, and the id also stays stable across re-renders and HTMX out-of-band swaps. This matters because the inline `oninput` handlers and the swatch `onclick` handlers find their target with `document.getElementById`: with a duplicate id, the swatches of the second picker would repaint the first one.

Hex Text Box

What it does: When *ShowValueInput* is True a narrow text box is rendered next to the picker and the two are kept in sync in both directions: dragging the picker writes the new value into the text box, and typing in the text box writes it back into the picker. Only a complete `#rrggbb` value is pushed back from the text box, so partially typed text is simply ignored until it is valid.

Property	Type	Default	Description
ShowValueInput	Boolean	False	Adds a text box showing the hex value, synchronized with the picker in both directions.

Swatches

What it does: *Swatches* is a list of colors, one `#rrggbb` value per line. Each one renders as a small square button under the picker; clicking it sets the picker (and the hex text box, when it is shown) to that color. Blank lines are skipped.

```
oPicker->Swatches->Text = "#0d6efd\r\n#198754\r\n#dc3545\r\n#ffc107";
```

Output

The inherited **HTML** property returns a `<div>` containing the optional label, the color input, the optional hex text box and the optional row of swatch buttons. The id of that `<div>` is **ColorPickerID** (or the FieldName-derived / auto id described above); the color input is `<id>_input` and the hex text box `<id>_text`. Serve the string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

For a one-off picker you do not need to keep an instance around: the class function **Build** creates a temporary picker, returns its markup and frees it. Its `aValue` argument is assigned unconditionally, so omitting it gives the `#0d6efd` default while passing an empty string on purpose renders `value=""`. This component is a plain form control: it is not data-aware and has no LoadFromDataSet method.

Reference

- Properties — 7 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_SignaturePad

TsgcHTMLComponent_SignaturePad captures a handwritten signature in an HTML5 <canvas> and exports it as a PNG data URL, ready to submit through a form post, an UploadURL fetch or a WebSocket message.

Introduction

TsgcHTMLComponent_SignaturePad emits a <canvas> and a small inline script that draws a stroke while the user drags a mouse, a finger or a stylus across it. The script listens to pointer events, so all three input kinds share a single code path, and it redraws the canvas at the device pixel ratio so the stroke stays sharp on high-DPI screens.

Every stroke is kept in an array, which is what makes the Undo button possible: it pops the last stroke and redraws the remaining ones. At the end of every stroke the canvas is exported with `toDataURL("image/png")` into a hidden input named after **FieldName**, so the current signature is always available as a PNG data URL, ready to be submitted.

Quick Start

1. Drop a **TsgcHTMLComponent_SignaturePad** component on the form (or create it at runtime).

2. Set **Width** and **Height** and, optionally, **PenColor**, **PenWidth** and a **PadID** of your own (leave PadID empty and a unique id is generated for you).

3. Read the **HTML** property and serve it inside a <form>, so the hidden **FieldName** input is posted with the rest of the fields.

```
TsgcHTMLComponent_SignaturePad *oPad = new TsgcHTMLComponent_SignaturePad(NULL);
try
{
  oPad->PadID = "employeeSignature";
  oPad->Width = 480;
  oPad->Height = 220;
  oPad->PenColor = "#1a1a2e";
  oPad->FieldName = "signature";
  html = oPad->HTML;
}
__finally
{
  delete oPad;
}
```

Canvas and Pen

What it does: *PadID* names the root and, from it, every generated element id (the canvas is `<PadID>_canvas`, the hidden input `<PadID>_input`, and the buttons `<PadID>_clear`, `_undo` and `_save`), all of which the inline script looks up with `document.getElementById`. *Width* and *Height* size the canvas in pixels. *PenColor* and *PenWidth* style the stroke, and *BackgroundColor* fills the canvas, so it is baked into the exported PNG.

Leave *PadID* empty (the default) and a unique element id is generated automatically the first time the component renders, and reused for the life of the instance, so two pads on the same page never collide (with a duplicate id, the second pad's script would draw on the first pad's canvas) and the id stays stable across re-renders and HTMX out-of-band swaps. Set it explicitly if you need a stable, known id, for example to reach the canvas or the hidden input

from your own JavaScript. Note that, unlike the other sgcHTML form components, the pad does not derive an id from *FieldName*: FieldName has a non-empty default, so two default pads would collide again.

Property	Type	Default	Description
PadID	string	(empty)	Root element id; also the prefix of the canvas, hidden input and button ids. Empty means a unique auto id, generated once per instance.
Width	Integer	400	Canvas width in pixels. The canvas is also styled max-width:100%.
Height	Integer	200	Canvas height in pixels.
PenColor	string	'#1a1a2e'	Stroke color used while drawing.
PenWidth	Integer	2	Stroke width in pixels. Lines use round caps and round joins.
BackgroundColor	string	'#ffffff'	Fills the canvas and is baked into the exported PNG, so a transparent signature is not possible.

Buttons and Localized Captions

What it does: *ShowClear* and *ShowUndo* toggle the Clear and Undo buttons; Clear drops every stroke, Undo drops only the last one. *ClearCaption*, *UndoCaption* and *SaveCaption* override the button text; each one defaults to an empty string in the constructor and is resolved at render time, falling back to the active sgcHTML locale (`sgcHTMLDefaultLocale.ClearText` / `.UndoText` / `.SaveText`, for example "Clear" / "Undo" / "Save" in English, "Borrar" / "Deshacer" / "Guardar" in Spanish, and so on for every supported language) when left empty. Resolving late, rather than in the constructor, is deliberate: a single component instance can render concurrent requests in different languages, and a value you set yourself always wins over the locale.

Property	Type	Default	Description
ShowClear	Boolean	True	Shows the Clear button, which drops every stroke.
ClearCaption	string	(locale)	Caption of the Clear button.
ShowUndo	Boolean	True	Shows the Undo button, which drops the last stroke.
UndoCaption	string	(locale)	Caption of the Undo button.
SaveCaption	string	(locale)	Caption of the Save button. Only rendered when UploadURL is set or WSSend is True.

Submitting the Signature

What it does: There are three ways to get the signature to the server. With a plain form, the pad renders a hidden input named *FieldName* inside the surrounding <form>, so a normal POST carries the PNG data URL, no extra property needed. Setting *UploadURL* makes a Save button appear; clicking it `fetch()`es a POST to that URL with `Content-Type: application/x-www-form-urlencoded` and a body of `<FieldName>=<urlencoded data URL>`. Setting *WSSend* to True wraps the whole pad in a `<form data-sgc-ws-send>` that also carries a hidden `action=signatureSave` field, and turns the Save button into a submit button; the sgcHTMX / WebSocket layer then serializes the form fields and sends them over the WebSocket connection instead of posting over HTTP, exactly like the Chat and Form components. If both UploadURL and WSSend are set, the button is a submit button and it also performs the fetch() upload. If FieldName is left empty, no hidden input is rendered and the signature cannot be submitted by any of the three paths.

Property	Type	Default	Description
FieldName	string	'signature'	Name of the hidden input carrying the PNG data URL. Empty renders no hidden input.
UploadURL	string	''	Sanitized target URL. Setting it shows the Save button and enables the fetch() upload.
WSSend	Boolean	False	Wraps the pad in a data-sgc-ws-send form and sends the fields over the WebSocket connection on Save.

Example of the WSSend path, where a connected **TsgcHTMX_Engine_Server** receives the serialized fields as a WebSocket message keyed by `action=signatureSave`:

```
TsgcHTMLComponent_SignaturePad *oPad = new TsgcHTMLComponent_SignaturePad(NULL);
try
{
  oPad->PadID = "contractSignature";
  oPad->FieldName = "signature";
  oPad->WSSend = true;
  html = oPad->HTML;
}
__finally
{
  delete oPad;
}
```

Example of the UploadURL path, where the Save button POSTs the data URL directly to a server endpoint:

```
TsgcHTMLComponent_SignaturePad *oPad = new TsgcHTMLComponent_SignaturePad(NULL);
try
{
  oPad->PadID = "deliverySignature";
  oPad->FieldName = "signature";
  oPad->UploadURL = "/api/signatures/save";
  html = oPad->HTML;
}
__finally
{
  delete oPad;
}
```

Read-Only Display

What it does: When *ReadOnly* is True the component renders no canvas and no script at all, which makes it suitable for a review page or a printable document. If *SignatureDataURL* holds a data URL (or any sanitized URL), it is drawn as an <img> at **Width** x **Height**; if it is empty, an empty framed placeholder box of the same size is drawn instead.

Property	Type	Default	Description
ReadOnly	Boolean	False	Renders a static <img> (or an empty placeholder) instead of an editable canvas.
SignatureDataURL	string	''	The signature to display in ReadOnly mode. Ignored unless ReadOnly is True. The URL is sanitized.

Output

The inherited **HTML** property returns the <canvas>, the hidden input, the button bar and the inline pointer-event script (or, in ReadOnly mode, just the <img> / placeholder markup, with no script). The component also contributes CSS through its **GetCSS** override: `.sgc-signature-pad .sgc-signature-actions` lays out the buttons in a flex row with a small gap, and `.sgc-signature-pad canvas` sets `touch-action:none` so a finger drag draws on the canvas instead of scrolling the page. Serve the HTML string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 16 properties available on the component.
- Methods — this component has no public methods.
- Events — this component has no published events.

TsgcHTMLComponent_NumPad

TsgcHTMLComponent_NumPad renders a touch-friendly, calculator-style numeric or currency keypad for point-of-sale entry, and keeps every value it emits locale-independent.

Introduction

TsgcHTMLComponent_NumPad lays out its keys in the familiar calculator order, 7 8 9 on the top row, 4 5 6, 1 2 3, then the decimal point, 0 and backspace on the last row, followed by a Clear and an Enter/Confirm button. A live numeric display sits above the keys, and a hidden `<input type="hidden">` named by **FieldName** carries the value that the surrounding form actually submits.

The pad is also a real keyboard control: it renders with `tabindex="0"`, so it can receive focus, and once focused it handles the digits 0-9, the decimal point, Backspace, Delete, Escape and Enter directly from the keyboard, not just from clicks on the buttons. See **Keyboard Accessibility** below for how it keeps two pads on the same page from reacting to each other's keystrokes.

Quick Start

1. Drop a **TsgcHTMLComponent_NumPad** component on the form (or create it at runtime).

2. Set **Mode**, **DecimalPlaces** and, for currency entry, **CurrencySymbol**; set **FieldName** to the name the hidden field should post under.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_NumPad *oNumPad = new TsgcHTMLComponent_NumPad(NULL);
try
{
  oNumPad->FieldName = "total";
  oNumPad->Mode = npCurrency;
  oNumPad->DecimalPlaces = 2;
  oNumPad->CurrencySymbol = "$";
  html = oNumPad->HTML;
}
__finally
{
  delete oNumPad;
}
```

Identity and Value

What it does: *PadID* is the id of the generated root element; *FieldName* names the hidden field the form posts; *Value* is the number the pad opens with.

Property	Type	Default	Description
PadID	string	''	id of the generated root element. Empty means a unique id, generated once per instance.

FieldName	string	'amount'	name attribute of the hidden field that carries the value. Empty renders no hidden field at all.
Value	Double	0	The number the pad opens with, shown in the display and in the hidden field.

Leave **PadID** empty, the default, and a unique id is generated the first time the pad renders and reused for the life of the instance, so two default-constructed pads on the same page never collide. Set it explicitly only when you need a stable, known id, for example to reach the pad from your own script.

Mode, Decimals and Currency

What it does: *Mode* chooses between whole numbers, decimals and currency; *DecimalPlaces* sets how many decimals are kept; *CurrencySymbol* prefixes the display and the quick-amount buttons under npCurrency.

Property	Type	Default	Description
Mode	TsgcHTMLNumPadMode	npDecimal	npInteger never renders the decimal-point key; npDecimal shows DecimalPlaces decimals; npCurrency shows DecimalPlaces decimals prefixed by CurrencySymbol.
DecimalPlaces	Integer	2	Number of decimals kept and displayed. Clamped internally to 0-6, and forced to 0 whenever Mode is npInteger.
CurrencySymbol	string	''	Text prefixed to the display and to each quick-amount button when Mode is npCurrency.

Locale-independent by construction. Every number this component writes, into the markup and into its own inline script, goes through `sgcHTMLFloatToStr`, never `FloatToStr` or `FormatFloat`. That is not a style preference. `FormatFloat` honors the thread locale, and on a server whose locale uses ',' as the decimal separator, formatting 12.5 that way and dropping the result straight into a JavaScript number literal turns one function argument into two, for example a call meant to read `amt(12.50)` becomes `amt(12,50)`, which is not the same call at all. The hidden field named by **FieldName** (default 'amount') always carries the canonical string '12.50', never '12,50', so the same form posts identical bytes whether the browser is Spanish, German or English.

Which Keys Appear

What it does: *ShowDisplay*, *ShowClear*, *ShowBackspace*, *ShowDecimalPoint* and *ShowEnter* toggle individual parts of the pad; *EnterCaption* and *ClearCaption* override the button text.

Property	Type	Default	Description
ShowDisplay	Boolean	True	Shows the running numeric display above the keys.
ShowClear	Boolean	True	Shows the Clear button.

ShowBack-space	Boolean	True	Shows the backspace key on the last row.
ShowDecimal-Point	Boolean	True	Shows the decimal-point key. Only ever rendered when Mode is not npInteger.
ShowEnter	Boolean	True	Shows the Enter/Confirm button.
EnterCaption	string	''	Text of the Enter button. Empty resolves from the active locale at render time.
ClearCaption	string	''	Text of the Clear button. Empty resolves from the active locale at render time.

Turning off **ShowBackspace** or **ShowDecimalPoint** does not collapse the grid: an invisible spacer takes the key's place, so the 0 key always stays under the 2, whichever keys are switched off. **EnterCaption** and **ClearCaption** are resolved when the pad renders, not when the property is read, and only when left empty; because the resolution happens at render time rather than at construction, a single component instance serving concurrent requests in different languages shows each request its own locale's Confirm and Clear text.

Limits, Size and Quick Amounts

What it does: *MaxValue* caps how large the entered value can grow; *ButtonSize* sets the pixel size of each key; *ColorStyle* picks the Bootstrap button variant for the digit keys; *QuickAmounts* renders one tender-shortcut button per entry.

Property	Type	Default	Description
MaxValue	Double	0	Ceiling on the entered value. 0 means no ceiling.
ButtonSize	Integer	64	Pixel size of each square key. Clamped internally to 28-160.
ColorStyle	TsgcHTML-ButtonStyle	bsOutli-neSecondary	Bootstrap button variant applied to the digit and decimal-point keys.
Quick-Amounts	TStrings	(empty)	One number per line, rendered as tender-shortcut buttons above the keys.

MaxValue is enforced in two different ways depending on how the value gets there. A digit typed (or clicked) that would push the buffer past MaxValue is rejected outright, the keystroke is simply dropped and the buffer stays as it was. Clicking a **QuickAmounts** button, by contrast, does not reject the amount, it clamps the entered value down to MaxValue instead. **ButtonSize** sets the width and height of every key; the pad's own width follows from it automatically, three key columns plus the gaps between them.

Every line in **QuickAmounts** is trimmed and parsed as a number before it is used. A blank line, or a line that is not a well-formed number, is silently dropped rather than reaching the markup. A line that does parse is re-emitted through the same locale-independent formatting as everything else in this component, so the button's caption and the value it sends both follow the pad's current Mode, DecimalPlaces and CurrencySymbol, never whatever raw text was typed into the collection. **ColorStyle** only affects the digit and decimal-point keys; Clear and Enter keep their own fixed Bootstrap styles regardless of ColorStyle.

```
oNumPad->QuickAmounts->Add("5");
oNumPad->QuickAmounts->Add("10");
oNumPad->QuickAmounts->Add("20");
oNumPad->QuickAmounts->Add("50");
```

Actions on Enter

What it does: *TargetSelector* mirrors the value into other elements of the page on Enter; *Action* POSTs the field to a URL when Enter is pressed; *WSSend* sends it over the WebSocket/HTMX transport instead.

Property	Type	Default	Description
TargetSelector	string	''	CSS selector of the element(s) that receive the value on Enter.
Action	string	''	When set (and WSSend is False), Enter POSTs the field to this URL.
WSSend	Boolean	False	When True, Enter sends the field over the WebSocket/HTMX transport instead of a plain POST.

Pressing Enter, by click or from the keyboard, first normalizes the buffer to the current value at DecimalPlaces precision and refreshes the display and the hidden field. If **TargetSelector** names a CSS selector, every matching element is updated too: an element exposing a `value` property receives the same fixed-decimal string and then has `input` and `change` DOM events dispatched on it (bubbling), so a framework listening for those events picks the change up exactly as if the user had typed it directly into that element; an element without a `value` property has its text content replaced instead, including the currency prefix under npCurrency.

If **WSSend** is True, or **Action** is non-empty, the Enter button is rendered as a real `type="submit"` button, and the pad's content is wrapped in a `<form>` so the browser's own submission handles delivery. WSSend wraps it in a `data-sgc-ws-send` form, the same mechanism **TsgcHTMLComponent_SignaturePad** and **TsgcHTMLComponent_Form** use: the WebSocket/HTMX layer serializes and sends the form's named fields. Action wraps it in a `method="POST"` form pointed at the sanitized URL, and the browser posts to it natively. When neither is set, Enter is a plain `type="button"` and nothing is submitted automatically. In every case, a `sgc:numpad CustomEvent` is dispatched on the pad's root element on Enter, carrying the pad's id and value in its detail, so page script can react client-side regardless of which submission path, if any, is active.

Keyboard Accessibility

The pad root renders with `tabindex="0"` and `role="group"`, so it behaves like a native form control for focus and tabbing. Once it has focus, its keydown handler responds to the digits 0-9, the decimal point (both '.' and ',' are accepted, so a keyboard that types a comma still works), Backspace, Delete, Escape (Delete and Escape both clear the buffer) and Enter.

The listener is attached to the pad's own root element, not to the page's document. That is a deliberate correctness choice: with a document-level listener, two independent NumPad instances on the same page, for example a tender pad and a change-due pad, would both react to every keystroke typed anywhere on the page. Scoping the listener to each pad's own element means only the pad that currently has focus responds, so multiple pads coexist on one page without cross-talk.

Output

The inherited **HTML** property returns the pad's `<div class="sgc-numpad">`, or the `<form>` wrapping it when **WSSend** or **Action** is set, containing the optional display, the optional quick-amount buttons, the key grid, the Clear/Enter actions row, and an inline `<script>` that wires the click and keydown handlers. Add the returned markup to the page; nothing further is required afterward. Serve it directly, or assign it to the BodyContent of a page template.

Reference

- Properties — 20 properties available on the component.
- Methods — this component has no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_CameraScanner

TsgcHTMLComponent_CameraScanner renders a live camera panel that scans barcodes and QR codes with the browser's native BarcodeDetector API and can also capture a still photo, with manual text entry as an always available fallback.

Introduction

TsgcHTMLComponent_CameraScanner opens the device camera with `getUserMedia` and shows the live video feed inside a bordered stage. Depending on **Mode**, it decodes barcodes and QR codes from the video frames, captures a still photo of the current frame, or both. Every result, whether it comes from a decoded barcode, a captured photo, or the manual entry field, is written to hidden form fields, optionally mirrored into another element on the page, optionally pushed to the server over the sgcHTMX WebSocket bridge or posted with `fetch()`, and always raised as a DOM `CustomEvent` so a page can react to it locally.

Native BarcodeDetector, no bundled decoder. Barcode and QR decoding is done entirely by the browser's native `window.BarcodeDetector` API. The component does not bundle a JavaScript decoding library and does not pull one from a Content Delivery Network, in keeping with the rest of this library, which ships self-contained assets and has to keep working in offline demos and air-gapped deployments. When `BarcodeDetector` is not available in the browser, the panel displays the message "Barcode scanning is not available in this browser, please type the code below." and keeps working: manual entry and photo capture do not depend on `BarcodeDetector` at all, so an operator can still get a value into the form. The camera itself, `getUserMedia`, is a much older and more widely supported API than `BarcodeDetector`, so this degradation path is the common case on browsers that do not yet ship a native decoder, not an edge case, most notably Safari and every browser on iPhone and iPad. Setting **PolyfillURL** to a barcode-detector polyfill script has the component load it on first use on any browser missing the native API, so scanning still works there instead of falling back to manual entry.

The camera always turns off. A camera panel that forgets to release its stream leaves the camera indicator light on after the operator has moved away from the page, which reads as a privacy problem even when nothing is actually being captured any more. The generated script stops every track of the active stream, and with it the indicator light, on three independent triggers: the `pagehide` event, the `beforeunload` event, and the tab becoming hidden through `visibilitychange`. Whichever way the operator leaves or backgrounds the page, the camera is released without waiting for a script on the next page to clean it up.

Quick Start

1. Drop a **TsgcHTMLComponent_CameraScanner** component on the form (or create it at runtime).

2. Set **Mode**, **FacingMode** and **FieldName**, and optionally **Action** or **WSSend** so a scan reaches the server.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_CameraScanner *oScanner = new TsgcHTMLComponent_CameraScanner(NULL);
try
{
  oScanner->ScannerID = "inboundScan";
  oScanner->Mode = smBoth;
  oScanner->FacingMode = sfEnvironment;
  oScanner->FieldName = "trackingCode";
  oScanner->Action = "/api/receiving/scan";
  html = oScanner->HTML;
}
__finally
{
  delete oScanner;
}
```

Identity & Sizing

What it does: *ScannerID* is the id of the generated root `<div>`; every other element the component emits (the video, the canvas, the hidden fields, the buttons) derives its id from it. *CSSWidth* and *CSSHeight* size the bordered stage that holds the video.

Property	Type	Default	Description
ScannerID	string	''	Root element id. Empty auto-generates and caches a unique id the first time the component renders.
CSSWidth	string	100%	CSS width of the video stage.
CSSHeight	string	320px	CSS height of the video stage.

Leave **ScannerID** empty (the default) and a unique id is generated automatically the first time the component renders, and reused for the life of the instance, so two default scanners on the same page never drive each other's `<video>` element. Set it explicitly for a stable, known id, for example to reach the panel from your own JavaScript.

Mode & Formats

What it does: *Mode* chooses whether the panel decodes barcodes, captures a photo, or both. *Formats* is the comma list of symbologies the decoder looks for. *PolyfillURL* loads a fallback decoder on browsers, such as Safari and iOS, that have no native BarcodeDetector. *FacingMode* picks the camera `getUserMedia` requests when no specific device has been chosen in the device selector.

Property	Type	Default	Description
Mode	TsgcHTMLScannerMode	smBoth	smBarcode decodes only, smPhoto captures only, smBoth does both.
Formats	string	qr_code,ean_13,code_128	Comma list of BarcodeDetector format strings.
PolyfillURL	string	''	Script that provides BarcodeDetector where the browser has none, e.g. iOS.
FacingMode	TsgcHTMLScannerFacing	sfEnvironment	sfEnvironment requests the rear camera, sfUser the front camera.

Formats is passed through untouched, as a quoted, script-escaped JSON array, to `new BarcodeDetector({formats: [...]})`; the component does not validate the tokens against a known list, it relies on the browser to accept or reject them. Empty entries in the comma list are dropped.

Scanning Behavior

What it does: *ContinuousScan* decides whether the camera keeps running after a successful decode. *ScanDelayMs* throttles how often a video frame is handed to the decoder. *BeepOnScan* plays an audible cue on every hit.

Property	Type	Default	Description
ContinuousScan	Boolean	True	Keeps decoding after a hit; False stops the camera on the first successful decode.
ScanDelayMs	Integer	400	Milliseconds between detection polls. Values below 50 are raised to 50.
BeepOnScan	Boolean	True	Plays a short WebAudio blip on every successful decode.

ScanDelayMs is a polling interval, not a debounce: while a detection call is in flight the next tick is skipped rather than queued, so a slow device never piles up overlapping detect() calls. Lowering the value makes the panel notice a code sooner at the cost of more CPU and battery use; raising it does the opposite.

Optional UI Controls

What it does: Each *Show** property toggles one piece of optional chrome around the video stage. The four caption properties override the English text shown on the buttons and the manual entry field, and are only consulted when non-empty.

Property	Type	Default	Description
ShowTorchButton	Boolean	True	Shows the torch button. It stays disabled unless the active camera track reports a torch capability.
ShowDeviceSelector	Boolean	True	Shows the camera picker, hidden automatically until more than one camera is found.
ShowManualEntry	Boolean	True	Shows the always available fallback text field and its submit button.
ShowCaptureButton	Boolean	True	Shows the capture button, only when Mode also includes photo capture.
ManualEntryPlaceholder	string	''	Placeholder of the manual entry field. Empty falls back to 'Enter code manually'.
StartCaption	string	''	Toggle button caption while the camera is off. Empty falls back to 'Start Camera'.
StopCaption	string	''	Toggle button caption while the camera is running. Empty falls back to 'Stop Camera'.

CaptureCaption	string	''	Capture button caption. Empty falls back to 'Capture Photo'.
TorchCaption	string	''	Torch button caption. Empty falls back to 'Torch'.

StartCaption and StopCaption both apply to the same button: it starts out reading StartCaption and switches to StopCaption the moment the stream is live, then back again on stop. ShowCaptureButton only puts the button on the page when Mode is smPhoto or smBoth; set Mode to smBarcode and the capture button disappears regardless of ShowCaptureButton, because there is no photo to capture. The manual entry submit button is not covered by any of the four caption properties: it always reads the current HTML locale's generic submit text.

Output & Actions

What it does: Every result the panel produces, a decoded barcode, a captured photo, or a manually typed value, goes through the same pipeline. It always fills the hidden fields named by **FieldName**, and it optionally mirrors the value into **TargetSelector**, pushes it to the server over WebSocket with **WSSend**, posts it to **Action**, or navigates the browser to **RedirectURL**; a DOM `CustomEvent` named `sgcCameraScanner:scan` is dispatched on every result no matter which of those are configured. **AutoStart** requests the camera as soon as the panel loads instead of waiting for the operator to click Start.

Property	Type	Default	Description
FieldName	string	scan	Name of the hidden input holding the value, and, with _photo appended, the one holding the captured JPEG.
Action	string	''	URL a result is POSTed to with fetch(). Empty performs no network request.
RedirectURL	string	''	Navigates the browser to this URL with the scanned value after a result.
TargetSelector	string	''	CSS selector of another page element to mirror the scanned value into.
WSSend	Boolean	False	Pushes every result to the server over the sgcHTMX WebSocket bridge.
AutoStart	Boolean	False	Requests the camera automatically when the panel loads.

The two hidden inputs (`<ScannerID>_input` for the value, `<ScannerID>_photo` for the photo) are always rendered, whatever FieldName holds, so the script can always find and fill them; only their `name` attribute, and therefore what an ordinary HTTP form post submits, depends on FieldName being non-empty. TargetSelector only ever receives the scanned or typed text value, never the captured photo. WSSend and Action are independent and can be used together: WSSend calls `window.sgcHTMX.send(...)` with the flat payload `{action:"cameraScan", scanner:"<id>", <FieldName>:value[, <FieldName>_photo:dataURL]}`, guarded so nothing breaks if the bridge script is not present on the page, while Action performs its own independent `fetch()` POST with the same two fields url-encoded in the body. The CustomEvent's `detail` carries that same payload, so a page can react purely client-side, with no server round trip, by listening for `sgcCameraScanner:scan` on `document`.

Output

The inherited **HTML** property returns a `<div class="sgc-camera-scanner">` containing the video stage, an offscreen canvas used to grab still frames, the hidden value and photo inputs, the optional device selector, toggle, torch and capture buttons, the optional manual entry field, a status line, and the inline script that drives all of it. When **WSSend** is True, the interior elements are wrapped in a `<form data-sgc-ws-send onsubmit="return false;">` that also carries a hidden `action=cameraScan` field, matching the markup shape used by the Chat and SignaturePad components; nothing in the panel actually submits that form, the WebSocket push happens by an explicit script call on every result instead. Add the returned markup to the page once, for example inside a card or a warehouse receiving form, and serve it directly or assign it to the BodyContent of a page template.

Reference

- Properties — 25 properties available on the component.
- Methods — this component has no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_Geolocation

TsgcHTMLComponent_Geolocation reads the position of the device with the browser Geolocation API and delivers it to the server, once per request or continuously, from a visible panel or from a hidden one.

Introduction

TsgcHTMLComponent_Geolocation asks the browser for the position of the device with `navigator.geolocation.getCurrentPosition` or `navigator.geolocation.watchPosition`. Every reading, and every error, is written to nine hidden form fields, optionally mirrored into another element of the page, optionally pushed to the server over the sgcHTMX WebSocket bridge or posted with `fetch()`, optionally followed by a redirect, and always raised as a DOM `CustomEvent` so a page can react to it locally.

Native API, no bundled library. The position comes from the browser Geolocation API. No JavaScript library is bundled and nothing is loaded from a Content Delivery Network, in keeping with the rest of this library, which ships self-contained assets.

Secure context required. Browsers only expose geolocation in a secure context: the page must be served over HTTPS, or from `localhost` while developing. On a plain HTTP page the panel does not ask for the position at all; it reports the `insecure_context` error instead. The first time a site asks for the position the browser also shows its own permission prompt, and a refusal is reported as `permission_denied`.

The watch always ends. A continuous watch is cleared when the Stop button is pressed, when page code calls `el.sgcGeoStop()`, when a permission denial arrives, and on the `pagehide` and `beforeunload` events, so the device does not keep tracking the position after the user has left the page.

Quick Start

1. Drop a **TsgcHTMLComponent_Geolocation** component on the form (or create it at runtime).

2. Choose **ShowUI** (visible panel or hidden) and **UpdateMode** (once or continuous), and set **WSSend** or **Action** so the position reaches the server.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent) from an HTTPS page.

```
TsgcHTMLComponent_Geolocation *oGeo = new TsgcHTMLComponent_Geolocation(NULL);
try
{
  oGeo->GeolocationID = "deliveryGeo";
  oGeo->ShowUI = false;
  oGeo->UpdateMode = gumContinuous;
  oGeo->MinIntervalMs = 30000;
  oGeo->MinDistanceMeters = 50;
  oGeo->AutoStart = true;
  oGeo->WSSend = true;
  html = oGeo->HTML;
}
__finally
{
  delete oGeo;
}
```

4. Handle the reading on the server. With **WSSend**, the message arrives in the OnHTMXMessage event of the engine assigned to the page:

With **Action** instead, the same nine fields arrive as an `application/x-www-form-urlencoded` POST body (`geo_lat=...&geo_lng=...`) in the route handler of that URL. DoLog and DoSavePosition above stand for your own code.

Visible or Hidden, Once or Continuous

What it does: *ShowUI* decides whether the user sees a panel. *UpdateMode* decides whether a start delivers one reading or keeps tracking. *AutoStart* starts the reading as soon as the panel loads.

Property	Type	Default	Description
GeolocationID	string	''	Root element id. Empty auto-generates and caches a unique id the first time the component renders.
ShowUI	Boolean	True	True renders the Start button (plus Stop in continuous mode), the status line and the optional map link. False renders only the hidden fields and the script.
ShowMapLink	Boolean	False	Adds an "Open in map" link to OpenStreetMap, shown after the first reading. Requires ShowUI.
UpdateMode	TsgcHTMLGeolocationUpdateMode	gumOnce	gumOnce delivers one reading per start, gumContinuous keeps watching until stopped.
AutoStart	Boolean	False	Starts the reading as soon as the panel loads.

The four combinations cover the usual cases:

ShowUI	UpdateMode	Typical use
True	gumOnce	A "Locate" button on a form, for example a check-in or a store finder. Each press delivers one reading.
True	gumContinuous	Start and Stop buttons that let the user decide when tracking runs, with the live position in the status line.
False	gumOnce	Silent capture of the position when the page loads (AutoStart = True), or when page code calls `el.sgcGeoStart()`.
False	gumContinuous	Background tracking of a delivery or field worker while the page stays open, throttled with MinIntervalMs and MinDistanceMeters.

A hidden panel has no button, so it must be started with **AutoStart** or from page code. The browser permission prompt is shown in every case.

Update Mode & Position Options

What it does: *MinIntervalMs* and *MinDistanceMeters* throttle the readings delivered in continuous mode. *EnableHighAccuracy*, *TimeoutMs* and *MaximumAgeMs* are passed to the browser as the PositionOptions of every request.

Property	Type	Default	Description
MinIntervalMs	Integer	5000	Continuous mode only. Minimum time between two delivered readings; 0 delivers every update.
MinDistanceMeters	Integer	0	Continuous mode only. Minimum distance from the last delivered reading; 0 disables the filter.
EnableHighAccuracy	Boolean	True	Asks the browser for the most accurate position it can provide.
TimeoutMs	Integer	10000	Time the browser may take to answer; 0 or less means no timeout.
MaximumAgeMs	Integer	0	Maximum age of a cached position the browser may return; 0 always asks for a fresh one.

In continuous mode the first reading after a start is always delivered. Each later browser update is delivered only when both thresholds are met. A timeout or an unavailable position keeps the watch running, while a permission denial ends it. In once mode any error ends the attempt.

Captions

Property	Type	Default	Description
StartCaption	string	''	Start button caption. Empty falls back to 'Locate'.
StopCaption	string	''	Stop button caption, continuous mode only. Empty falls back to 'Stop'.
LocatingCaption	string	''	Status line text while the browser is locating. Empty falls back to 'Locating...'.

Output & Actions

What it does: Every reading and every error goes through the same pipeline. It always fills the hidden fields named after **FieldName**, and it optionally mirrors the value into **TargetSelector**, pushes it over WebSocket with **WSSend**, posts it to **Action** and, after a successful reading, navigates to **RedirectURL**. A DOM `CustomEvent` named `sgcGeolocation:position` is dispatched on every reading no matter which of those are configured.

Property	Type	Default	Description
FieldName	string	geo	Prefix of the nine hidden field names and of the payload members.
Action	string	''	URL every reading or error is POSTed to with fetch(). Empty performs no network request.
RedirectURL	string	''	Navigates the browser to this URL, with the reading in the query string, after a successful reading.
TargetSelector	string	''	CSS selector of another page element that receives "lat,lng" or the error code.
WSSend	Boolean	False	Pushes every reading to the server over the sgcHTMX WebSocket bridge.

With the default FieldName the delivered fields are:

Field	Content
geo_lat	Latitude in decimal degrees.
geo_lng	Longitude in decimal degrees.
geo_accuracy	Accuracy of the latitude and longitude, in meters.
geo_altitude	Altitude in meters, empty when the device does not report it.
geo_altitude_accuracy	Accuracy of the altitude in meters, empty when not reported.
geo_heading	Direction of travel in degrees, empty when not reported.
geo_speed	Speed in meters per second, empty when not reported.
geo_timestamp	Time of the reading as reported by the browser, in milliseconds since the Unix epoch.
geo_error	Empty on success, otherwise one of the error codes below. On an error every other field is empty.

Error codes reported in `geo_error`:

Code	Meaning
permission_denied	The user or the browser refused access to the position. A continuous watch is ended.
position_unavailable	The browser could not determine the position. A continuous watch keeps running.

timeout	No position arrived within TimeoutMs. A continuous watch keeps running.
insecure_context	The page is not served over HTTPS or from localhost, so the browser does not expose geolocation.
not_supported	The browser has no Geolocation API, or the request could not be made.

The WebSocket payload, which is also the `detail` of the DOM event, is the flat object `{action:"geolocation", geolocation:"<GeolocationID>", geo_lat, geo_lng, ...geo_error}`. The Action POST sends the same nine fields url-encoded in the body. The redirect appends only the non empty position fields to the query string and never happens after an error; when Action is also set, it waits until the POST has settled. When FieldName is empty, the hidden inputs have no `name` and the payload uses the bare keys `lat`, `lng` and so on.

JavaScript Hooks

The script sets two functions on the root element, so page code can drive the panel, which is the only way to start a hidden panel without AutoStart:

```
var el = document.getElementById("deliveryGeo");
el.sgcGeoStart();   // same as pressing Start
el.sgcGeoStop();    // same as pressing Stop

document.addEventListener("sgcGeolocation:position", function (e) {
  if (!e.detail.geo_error)
    console.log(e.detail.geo_lat + ", " + e.detail.geo_lng);
});
```

Output

The inherited **HTML** property returns a `<div class="sgc-geolocation">` containing the nine hidden inputs (`<GeolocationID>_lat` to `<GeolocationID>_error`), and, when ShowUI is True, the buttons, the optional map link and a status line, followed by the inline script that drives all of it. When **WSSend** is True, the interior elements are wrapped in a `<form data-sgc-ws-send onsubmit="return false;">` that also carries a hidden `action=geolocation` field, the same markup shape used by the CameraScanner component; nothing in the panel submits that form, the WebSocket push happens by an explicit script call on every reading instead.

Reference

- Properties: 18 properties available on the component.
- Methods: this component has no public methods of its own.
- Events: this component has no published events.

TsgcHTMLComponent_Panel

TsgcHTMLComponent_Panel renders a Bootstrap card with an optional header, body and footer, that can be solid or outlined, collapsible, and vertically scrollable.

Introduction

TsgcHTMLComponent_Panel emits a Bootstrap `card` built from three parts: a header (**Title**), a body (**Body**) and a footer (**Footer**). Any part you leave empty is omitted. The card colour comes from **Color**, and **Outline** chooses between a solid `text-bg-*` card and a bordered `border-*` card. The complete markup is returned through the inherited **HTML** property.

Set **Collapsible** to wrap the body in a Bootstrap collapse that toggles when the header is clicked, with **Expanded** deciding the initial state. Set **Scrollable** together with **MaxHeight** to cap the body height and add a vertical scrollbar.

Quick Start

1. Drop a **TsgcHTMLComponent_Panel** component on the form (or create it at runtime).

2. Set the **Title** and **Body**, and pick a **Color**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Panel *oPanel = new TsgcHTMLComponent_Panel(NULL);
try
{
  oPanel->Title = "Welcome";
  oPanel->Body = "<p>Your dashboard is ready.</p>";
  oPanel->Color = hcLight;
  oPanel->Footer = "Updated just now";
  html = oPanel->HTML;
}
__finally
{
  delete oPanel;
}
```

Content

What it does: *Title* fills the card header, *Body* the card body and *Footer* the card footer. The body and footer accept raw HTML, so you can embed any markup. Leave a part empty to omit it.

Property	Type	Default	Description
Title	string	''	Header text. When empty no header is drawn. Plain text: markup in it is shown literally, while Body and Footer still accept raw HTML.

Body	string	''	Card body content (raw HTML allowed).
Footer	string	''	Footer content. When empty no footer is drawn.
PanelID	string	''	The id of the generated card and the base id for the collapse region.

Appearance

What it does: *Color* selects a Bootstrap contextual colour, and *Outline* switches between a solid card and a bordered (outlined) card.

Property	Type	Default	Description
Color	TsgcHTMLColor	hcLight	Card colour: hcPrimary, hcSecondary, hcSuccess, hcDanger, hcWarning, hcInfo, hcLight, hcDark, hcWhite or hcMuted.
Outline	Boolean	False	When True draws a bordered card (border-*) instead of a solid one (text-bg-*).

```
oPanel->Color = hcSuccess;
oPanel->Outline = true;
```

Collapsing and Scrolling

What it does: *Collapsible* makes the header toggle the body open and closed, with *Expanded* setting the initial state. *Scrollable* together with *MaxHeight* caps the body height and adds a vertical scrollbar for long content.

Property	Type	Default	Description
Collapsible	Boolean	False	When True the header toggles a Bootstrap collapse around the body.
Expanded	Boolean	True	Initial state of a collapsible panel; True shows the body, False hides it.
Scrollable	Boolean	False	When True (with MaxHeight set) the body becomes vertically scrollable.
MaxHeight	string	''	CSS max-height for a scrollable body, e.g. '240px'.

```
oPanel->Collapsible = true;
oPanel->Expanded = false;
oPanel->Scrollable = true;
oPanel->MaxHeight = "240px";
```

One-Line Build

When you just need the markup, the static **Build** class function creates a panel, fills in title, body, colour and footer, returns the HTML and frees the component for you.

```
html = TsgcHTMLComponent_Panel::Build("Status", "All systems go.", hcSuccess);
```

Output

The inherited **HTML** property returns the complete `<div class="card">` markup, including the header, collapse wrapper (when collapsible) and footer. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 10 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_StatCard

TsgcHTMLComponent_StatCard renders a compact dashboard metric card showing a title, a large value, an optional icon, a trend arrow and either a contextual colour or a gradient background.

Introduction

TsgcHTMLComponent_StatCard emits a Bootstrap `card` tuned for KPI tiles. It shows a small **Title**, a big **Value**, an optional **Icon** in the corner, a coloured **Trend** arrow with a **TrendValue**, and an optional **FooterText**. The card is styled either with a Bootstrap contextual **Color** or, for a richer look, with a **Gradient** background. The complete markup is returned through the inherited **HTML** property.

Use **GetInnerHTML** when you only need the inner fragment (subtitle, value and trend) to drop into a card you are building yourself, for example a cell of a custom grid.

Quick Start

1. Drop a **TsgcHTMLComponent_StatCard** component on the form (or create it at runtime).

2. Set the **Title** and **Value**, pick a **Color**, and optionally a **Trend**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_StatCard *oCard = new TsgcHTMLComponent_StatCard(NULL);
try
{
  oCard->Title = "Revenue";
  oCard->Value = "$48,250";
  oCard->Color = scSuccess;
  oCard->Trend = stUp;
  oCard->TrendValue = "12%";
  html = oCard->HTML;
}
__finally
{
  delete oCard;
}
```

Content

What it does: *Title* is the small caption, *Value* is the large figure, *Icon* is optional markup shown in the top-right corner, and *FooterText* is small print under the value. The icon accepts any HTML, so you can pass an icon-font element or an emoji.

Property	Type	Default	Description
Title	string	''	Small caption shown above the value.
Value	string	''	The large metric value.

Icon	string	''	Optional icon markup shown in the top-right corner.
FooterText	string	''	Small text drawn in the card footer.
CardID	string	''	The id of the generated card element.
CSSClass	string	''	Extra CSS class names appended to the card.

Trend

What it does: *Trend* selects an arrow (up, down or neutral) and its colour, and *TrendValue* is the text shown next to the arrow (for example a percentage). The trend appears in the footer alongside any *FooterText*.

Property	Type	Default	Description
Trend	TsgcHTMLStatTrend	stNone	Trend arrow: stNone, stUp, stDown or stNeutral.
TrendValue	string	''	Text shown next to the trend arrow, e.g. '12%'.

```
oCard->Trend = stDown;
oCard->TrendValue = "3%";
```

Appearance

What it does: *Color* selects a Bootstrap contextual colour for the border, icon and value text. For a bolder tile, *Gradient* applies a full gradient background; you can pick a preset or supply *GradientStartColor* and *GradientEndColor*, and tune the *GradientAngle* and *GradientBorderRadius*.

Property	Type	Default	Description
Color	TsgcHTMLStatColor	scPrimary	Contextual colour: scPrimary, scSecondary, scSuccess, scDanger, scWarning, scInfo, scDark or scLight.
Gradient	TsgcHTMLStatGradient	sgNone	Gradient preset: sgNone, sgBlueViolet, sgPinkRed, sgBlueAqua or sgGreenTeal.
GradientStartColor	string	''	Start colour of a custom gradient (used when Gradient is sgNone).
GradientEndColor	string	''	End colour of a custom gradient (used when Gradient is sgNone).

GradientAngle	Integer	135	Gradient direction in degrees.
GradientBorderRadius	string	'12px'	Corner radius applied to a gradient card.

```
oCard->Gradient = sgBlueViolet;
oCard->GradientAngle = 120;
```

One-Line Build

When you just need the markup, the static **Build** class function creates a card, fills in the common fields and returns the HTML, freeing the component for you.

```
html = TsgcHTMLComponent_StatCard::Build("Users", "1,204", scInfo, stUp, "8%");
```

Output

The inherited **HTML** property returns the complete `<div class="card">` markup including body and footer. Serve this string directly, place several cards in a Bootstrap grid for a dashboard row, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 14 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Accordion

TsgcHTMLComponent_Accordion renders a Bootstrap accordion: a stack of collapsible panels, each with a header and a body, that expand and collapse when their header is clicked.

Introduction

TsgcHTMLComponent_Accordion emits a Bootstrap `accordion` made of one or more items. Each item is a **TsgcHTMLAccordionItem** with a *Title* (the clickable header), a *Content* (the body) and an *Expanded* flag for its initial state. You build the panel list through the **Items** collection, then read the inherited **HTML** property for the complete markup.

By default only one item is open at a time, like a true accordion, because every panel points at the same parent container. Set **AlwaysOpen** to let several panels stay open independently. Set **Flush** for an edge-to-edge style with no outer borders or rounded corners.

Quick Start

1. Drop a **TsgcHTMLComponent_Accordion** component on the form (or create it at runtime).

2. Add one or more **Items**, each with a *Title* and *Content*.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Accordion *oAccordion = new TsgcHTMLComponent_Accordion(NULL);
try
{
  oAccordion->AccordionID = "faq";
  TsgcHTMLAccordionItem *oItem = oAccordion->Items->Add();
  oItem->Title = "What is sgcHTML?";
  oItem->Content = "<p>A server-side HTML component library.</p>";
  oItem->Expanded = true;
  oItem = oAccordion->Items->Add();
  oItem->Title = "Does it need a CDN?";
  oItem->Content = "<p>No, assets are served locally.</p>";
  html = oAccordion->HTML;
}
__finally
{
  delete oAccordion;
}
```

Items

What it does: The *Items* collection holds the panels. Each **TsgcHTMLAccordionItem** exposes *Title* (the header text), *Content* (the body markup, raw HTML allowed) and *Expanded* (whether that panel starts open).

Property	Type	Default	Description

Items	TsgcHTMLAccordion-Items	(empty)	Collection of TsgcHTMLAccordionItem panels.
Items[i].Title	string	''	Header text of the panel (the clickable toggle).
Items[i].Content	string	''	Body content of the panel (raw HTML allowed).
Items[i].Expanded	Boolean	False	When True the panel is open in the initial render.

Behaviour and Style

What it does: *AccordionID* is the id shared by the panels and used to coordinate single-open behaviour. *AlwaysOpen* lets several panels stay open at once. *Flush* removes the outer borders and rounded corners for an edge-to-edge look.

Property	Type	Default	Description
Accordion-ID	string	'sgcAccordion'	The id of the accordion container, shared by its panels.
Flush	Boolean	False	Edge-to-edge style with no outer borders or rounded corners.
AlwaysOpen	Boolean	False	When True several panels can stay open at once.

```
oAccordion->Flush = true;
oAccordion->AlwaysOpen = true;
```

Output

The inherited **HTML** property returns the complete `<div class="accordion">` markup with one `accordion-item` per panel. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document. When several accordions share a page, give each a unique **AccordionID**.

Reference

- Properties — 4 properties available on the component.
- Methods — this component has no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_Carousel

TsgcHTMLComponent_Carousel renders a Bootstrap carousel: a rotating slideshow of images, each with an optional caption and description, with indicators, previous/next controls and optional autoplay.

Introduction

TsgcHTMLComponent_Carousel emits a Bootstrap `carousel` built from a list of slides. Each slide is a **TsgcHTMLCarouselItem** with an *ImageSrc*, an *Alt* text, and an optional *Caption* and *Description*. You build the list through the **Items** collection or with the **AddSlide** helper, then read the inherited **HTML** property for the complete markup.

Indicators (the dots), previous/next controls, autoplay, the slide **Interval**, a slide or fade **Transition**, a **Dark** variant and a fixed **Height** are all controlled by properties. The first slide is active by default.

Quick Start

1. Drop a **TsgcHTMLComponent_Carousel** component on the form (or create it at runtime).

2. Add a few slides with **AddSlide** (or through the **Items** collection).

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Carousel *oCarousel = new TsgcHTMLComponent_Carousel(NULL);
try
{
  oCarousel->CarouselID = "hero";
  oCarousel->AddSlide("/img/slide1.jpg", "Fast", "Built for speed.");
  oCarousel->AddSlide("/img/slide2.jpg", "Secure", "TLS everywhere.");
  html = oCarousel->HTML;
}
__finally
{
  delete oCarousel;
}
```

Slides

What it does: The *Items* collection holds the slides. Each **TsgcHTMLCarouselItem** exposes *ImageSrc* (the picture), *Alt* (accessible text), *Caption* and *Description* (overlaid text) and *Active* (which slide shows first). The **AddSlide** method appends a slide and marks the first one active for you.

Property	Type	Default	Description
Items	TsgcHTML-CarouselItems	(empty)	Collection of TsgcHTMLCarouselItem slides.
Items[i].ImageSrc	string	''	URL or path of the slide image.

Items[i].Alt	string	''	Alternative text for the image.
Items[i].Caption	string	''	Heading overlaid on the slide.
Items[i].Description	string	''	Paragraph overlaid under the caption.
Items[i].Active	Boolean	False	Marks this slide as the one shown first.

Navigation

What it does: *ShowIndicators* draws the dot indicators, *ShowControls* draws the previous/next arrows, and *CarouselID* is the id that ties the indicators and controls to the slides.

Property	Type	Default	Description
CarouselID	string	'sgc-Carousel'	The id of the carousel container, shared by indicators and controls.
ShowIndicators	Boolean	True	Shows the slide indicator dots.
ShowControls	Boolean	True	Shows the previous and next arrow controls.

Playback and Appearance

What it does: *Autoplay* rotates the slides automatically, *Interval* sets the time per slide, *Transition* chooses slide or fade, *Dark* switches to the dark variant and *Height* caps the slide height.

Property	Type	Default	Description
Autoplay	Boolean	True	Rotates the slides automatically.
Interval	Integer	5000	Time each slide is shown, in milliseconds.
Transition	TsgcHTMLCarouselTransition	ctSlide	Transition style: ctSlide or ctFade.
Dark	Boolean	False	Uses the dark carousel variant (dark controls and captions).
Height	string	''	Fixed slide height (CSS length); images are cropped with object-fit cover.

```
oCarousel->Transition = ctFade;
oCarousel->Interval = 3000;
oCarousel->Height = "420px";
```

Output

The inherited **HTML** property returns the complete `<div class="carousel slide">` markup with indicators, an `carousel-inner` holding one `carousel-item` per slide, and the controls. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document. When several carousels share a page, give each a unique **CarouselID**.

Reference

- Properties — 9 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Image

TsgcHTMLComponent_Image emits a responsive Bootstrap `<img>` element with an optional shape, lazy loading, lightbox link and caption.

Introduction

TsgcHTMLComponent_Image generates the markup for a single image. You set the **Src** (and optionally **Alt** text), pick a **Shape**, and the component returns the complete `<img>` tag through its inherited **HTML** property. When **Responsive** is on, the image is given the Bootstrap `img-fluid` class so it scales with its container. When a **Caption** is set, the image is wrapped in a `<figure>` with a `<figcaption>`.

For a quick one-line render without creating an instance, use the static **Build** method.

Quick Start

1. Drop a **TsgcHTMLComponent_Image** component on the form (or create it at runtime).

2. Set the **Src**, an **Alt** text and, if you like, a **Shape**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Image *oImage = new TsgcHTMLComponent_Image(NULL);
try
{
  oImage->Src = "/img/team.jpg";
  oImage->Alt = "Our team";
  oImage->Shape = isRounded;
  oImage->Caption = "The team at work";
  html = oImage->HTML;
}
__finally
{
  delete oImage;
}
```

Source and Appearance

What it does: *Src* is the image URL and *Alt* the alternative text. *Shape* selects a Bootstrap shape (rounded corners, a circle or a framed thumbnail). *Width* and *Height* set explicit dimensions, while *Responsive* lets the image scale with its container. *ImageID* assigns an `id` to the tag.

Property	Type	Default	Description
Src	string	''	Image URL written to the `src` attribute.
Alt	string	''	Alternative text for accessibility and broken-image fallback.

Shape	TsgcHTMLIm-ageShape	isNone	Bootstrap shape: isNone, isRounded, isCircle or isThumb-nail.
Width	string	''	Explicit width in pixels written to the `width` attribute.
Height	string	''	Explicit height in pixels written to the `height` attribute.
Respon-sive	Boolean	True	When True applies the `img-fluid` class so the image scales with its container.
ImageID	string	''	Optional `id` attribute for the `<img>` element.

Behaviour

What it does: *LazyLoad* defers loading until the image scrolls into view by emitting `loading="lazy"`. *Lightbox* wraps the image in a link that opens the full-size source in a new tab. *Caption* wraps the image in a `<figure>` with a visible caption.

Property	Type	Default	Description
LazyLoad	Boolean	True	Emits `loading="lazy"` so the browser defers loading off-screen images.
Lightbox	Boolean	False	Wraps the image in an anchor that opens the source in a new tab.
Caption	string	''	When set, wraps the image in a `<figure>` with this caption text.

```
// One-line render with the static Build helper
html = TsgcHTMLComponent_Image::Build("/img/logo.png", "Logo", isCircle);
```

Output

The inherited **HTML** property returns the `<img>` tag, optionally wrapped in an anchor (when **Lightbox** is on) and/or a `<figure>` (when **Caption** is set). Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 10 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Avatar

TsgcHTMLComponent_Avatar renders a user avatar from an image or initials, with a configurable size, shape and online status indicator.

Introduction

TsgcHTMLComponent_Avatar emits a small, self-contained block of HTML representing a person. When an **ImageURL** is set it shows that picture; otherwise it draws a colored circle with the supplied **Initials**. You choose a **Size** and a **Shape**, and optionally add a corner **Status** dot (online, offline, busy or away). The complete markup is returned through the inherited **HTML** property.

For a quick one-line render without creating an instance, use the static **Build** (initials) or **BuildImage** (picture) methods.

Quick Start

1. Drop a **TsgcHTMLComponent_Avatar** component on the form (or create it at runtime).

2. Set the **Initials** (or an **ImageURL**), a **Size** and, if you like, a **Status**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Avatar *oAvatar = new TsgcHTMLComponent_Avatar(NULL);
try
{
  oAvatar->Initials = "JS";
  oAvatar->Size = asLarge;
  oAvatar->Status = atOnline;
  html = oAvatar->HTML;
}
__finally
{
  delete oAvatar;
}
```

Content

What it does: *ImageURL* is the picture to show; when it is empty the avatar draws *Initials* on a colored background instead. *AltText* is the image alternative text. *AvatarID* assigns an `id` to the wrapper.

Property	Type	Default	Description
ImageURL	string	”	Picture to display. When empty, the initials are drawn instead.
Initials	string	”	One or two letters drawn when no image is set.
AltText	string	”	Alternative text for the avatar image.

AvatarID	string	''	Optional `id` attribute for the wrapping element.

Size and Shape

What it does: *Size* selects a fixed pixel size (32, 48, 64 or 96 px). *Shape* chooses a circle, rounded square or square. *Color* (a CSS color) and *ColorStyle* (a Bootstrap theme color) set the background behind the initials.

Property	Type	Default	Description
Size	TsgcHTMLAvatarSize	asMedium	Avatar size: asSmall (32px), asMedium (48px), asLarge (64px) or asXLarge (96px).
Shape	TsgcHTMLAvatarShape	apCircle	Avatar shape: apCircle, apRounded or apSquare.
Color	string	''	CSS color for the initials background. Overrides ColorStyle when set.
ColorStyle	TsgcHTMLColor	hcPrimary	Bootstrap theme color for the initials background.

Status Indicator

What it does: *Status* adds a small colored dot in the bottom-right corner to signal presence. *StatusBorderWidth* and *StatusBorderColor* control the ring drawn around that dot so it stands out against the avatar.

Property	Type	Default	Description
Status	TsgcHTMLAvatarStatus	atNone	Presence dot: atNone, atOnline, atOffline, atBusy or atAway.
StatusBorderWidth	Integer	2	Width in pixels of the ring around the status dot.
StatusBorderColor	TsgcHTMLColor	hcWhite	Bootstrap theme color of the ring around the status dot.

```
// One-line render with the static helpers
html = TsgcHTMLComponent_Avatar::Build("JS", asLarge, "#0d6efd", atOnline);
// or from a picture
html = TsgcHTMLComponent_Avatar::BuildImage("/img/jane.jpg", asLarge, atBusy);
```

Output

The inherited **HTML** property returns an inline-block `<div>` holding either the image or the initials block, plus the status dot when a status is set. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 11 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Video

TsgcHTMLComponent_Video renders a responsive HTML5 `<video>` or `<audio>` player, or an embedded YouTube frame.

Introduction

TsgcHTMLComponent_Video emits the markup for a media player. You set the **Src** and choose a **MediaType** (video or audio); the component returns the complete tag through its inherited **HTML** property. Playback flags such as **Controls**, **Autoplay**, **Loop** and **Muted** map directly to the corresponding HTML5 attributes. For video, a **Poster** frame and a responsive 16:9 wrapper are available.

For a quick one-line render without creating an instance, use the static **Build** (local file) or **BuildYouTube** (embedded YouTube) methods.

Quick Start

1. Drop a **TsgcHTMLComponent_Video** component on the form (or create it at runtime).

2. Set the **Src**, choose a **MediaType** and enable the playback flags you need.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Video *oVideo = new TsgcHTMLComponent_Video(NULL);
try
{
  oVideo->Src = "/media/intro.mp4";
  oVideo->Poster = "/media/intro.jpg";
  oVideo->Controls = true;
  html = oVideo->HTML;
}
__finally
{
  delete oVideo;
}
```

Source and Type

What it does: *Src* is the media URL and *MediaType* selects between a `<video>` and an `<audio>` element. *VideoID* assigns an `id` to the player.

Property	Type	Default	Description
Src	string	''	Media URL written to the `src` attribute.
MediaType	TsgcHTMLMediaType	mtVideo	Element kind: mtVideo (a <video> tag) or mtAudio (an <audio> tag).

VideoID	string	''	Optional `id` attribute for the player element.

Playback

What it does: *Controls* shows the browser's playback controls. *Autoplay*, *Loop* and *Muted* map to the matching HTML5 attributes (most browsers require *Muted* for autoplay to start). *Poster* is the still frame shown before a video plays.

Property	Type	Default	Description
Controls	Boolean	True	Shows the browser's native playback controls.
Autoplay	Boolean	False	Starts playback automatically (usually requires Muted).
Loop	Boolean	False	Restarts playback when the media ends.
Muted	Boolean	False	Starts the media muted.
Poster	string	''	Still image shown before a video plays (video only).

Layout

What it does: *Width* and *Height* set the player size through an inline style. *Responsive* wraps a video in a Bootstrap `ratio ratio-16x9` container so it scales with its column.

Property	Type	Default	Description
Width	string	'100%'	Player width (any CSS length, e.g. `100%` or `640px`).
Height	string	''	Player height (any CSS length).
Responsive	Boolean	True	Wraps a video in a 16:9 ratio container so it scales with its container.

```
// One-line render with the static helpers
html = TsgcHTMLComponent_Video::Build("/media/song.mp3", mtAudio, true);
// or embed a YouTube video
html = TsgcHTMLComponent_Video::BuildYouTube("dQw4w9WgXcQ");
```

Output

The inherited **HTML** property returns the `<video>` or `<audio>` tag, wrapped in a responsive 16:9 `<div>` when **Responsive** is on for a video. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 11 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Splitter

TsgcHTMLComponent_Splitter renders a drag-resizable two-pane layout, with a gutter the visitor drags to change the split between pane A and pane B.

Introduction

TsgcHTMLComponent_Splitter emits a flexbox root `<div>` holding three children: pane A, a draggable gutter and pane B. Depending on **Orientation** the panes sit side by side with a vertical gutter, or stacked with a horizontal gutter. A small inline script drives the drag using pointer events, so mouse, touch and pen input all go through the same code path, and it calls `setPointerCapture` so the drag keeps tracking even when the pointer leaves the gutter.

Each pane holds trusted raw markup, in the same way as **TsgcHTMLComponent_Panel.Body**, so its content is not HTML-encoded. Never feed unescaped visitor input into **PaneAContent**, **PaneBContent**, **AddPaneA** or **AddPaneB**.

Quick Start

1. Drop a **TsgcHTMLComponent_Splitter** component on the form (or create it at runtime).

2. Set **Orientation** and **InitialSplit**, then add markup to each pane with **AddPaneA** and **AddPaneB**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Splitter *oSplitter = new TsgcHTMLComponent_Splitter(NULL);
try
{
  oSplitter->Orientation = soHorizontal;
  oSplitter->InitialSplit = 30;
  oSplitter->Height = "600px";
  oSplitter->AddPaneA("<h3>Navigation</h3>");
  oSplitter->AddPaneB("<h3>Details</h3>");
  html = oSplitter->HTML;
}
__finally
{
  delete oSplitter;
}
```

Layout and Orientation

What it does: *Orientation* chooses whether the panes sit side by side or stacked. *Height* sets the height of the whole splitter, and *SplitterID* is the id of the root element, useful when a page hosts more than one splitter.

Property	Type	Default	Description

Orientation	TsgcHTMLSplitterOrientation	soHorizontal	soHorizontal places the panes side by side with a vertical, col-resize gutter. soVertical stacks the panes with a horizontal, row-resize gutter.
Height	string	'400px'	CSS length applied to the root element's height. An empty value falls back to '400px'.
SplitterID	string	(empty)	The id of the root element. Empty means a unique auto id, generated once per instance.

Leave *SplitterID* empty (the default) and a unique element id is generated automatically the first time the component renders, and reused for the life of the instance, so two splitters on the same page, including nested ones, never collide, and the id stays stable across re-renders and HTMX out-of-band swaps. This matters because the inline drag script finds its own root with `document.getElementById`: with a duplicate id, the second splitter's drag handler would resize the first splitter's panes. Set *SplitterID* explicitly if you need a stable, known id, for example to reach the element from your own JavaScript.

Sizing and Constraints

What it does: *InitialSplit* sets the starting share of pane A before any persisted value is restored. *MinSizeA* and *MinSizeB* keep each pane from being dragged smaller than a fixed pixel size. *GutterSize* controls the thickness of the drag bar between the panes.

Property	Type	Default	Description
InitialSplit	Integer	50	Percent of the root taken by pane A on first render, before any value persisted under PersistKey is restored.
MinSizeA	Integer	0	Minimum size of pane A, in pixels, enforced while dragging.
MinSizeB	Integer	0	Minimum size of pane B, in pixels, enforced while dragging.
GutterSize	Integer	6	Thickness of the drag bar, in pixels.

Pane Content

What it does: Each pane holds a block of trusted raw HTML. Set *PaneAContent* or *PaneBContent* to replace a pane's content in one assignment, or call **AddPaneA** and **AddPaneB** to append one line of markup at a time. Because the content is added with AddRaw, exactly like **TsgcHTMLComponent_Panel.Body**, it is never HTML-encoded.

Property	Type	Default	Description

PaneAContent	TStrings	(empty)	Raw HTML lines rendered inside the first pane.
PaneBContent	TStrings	(empty)	Raw HTML lines rendered inside the second pane.

```
oSplitter->AddPaneA("<p>Left pane content.</p>");
oSplitter->AddPaneB("<p>Right pane content.</p>");
```

Persisting the Split

What it does: *PersistKey* saves the current split to the browser's `window.localStorage` whenever the visitor releases the gutter, and restores it the next time the page loads.

Property	Type	Default	Description
PersistKey	string	''	window.localStorage key used to save and restore the split percentage. Empty means no persistence.

Persistence is per browser and per origin, and it survives page reloads until the visitor clears site data. *PersistKey* is independent of *SplitterID*: the element ids no longer collide on their own, but the localStorage key still does, so give every splitter on a page, nested ones included, its own distinct PersistKey.

Output

The inherited **HTML** property returns the root `<div class="sgc-splitter d-flex">`, containing a `<div class="sgc-splitter-pane">` for pane A, a `<div class="sgc-splitter-gutter">` and a second `<div class="sgc-splitter-pane">` for pane B, followed by the inline drag script. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document. A splitter can be nested inside a pane of another splitter: each one gets its own element id automatically, and only **PersistKey** has to be made distinct by hand.

Reference

- Properties — 10 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_PDFViewer

TsgcHTMLComponent_PDFViewer embeds a PDF viewer in the page, built on Mozilla pdf.js, with page navigation, zoom, search, download and print.

Introduction

TsgcHTMLComponent_PDFViewer emits a viewer panel: an optional toolbar, a `<canvas>` where the current page is drawn, an error area, and the script that loads the document with pdf.js and renders it. The document is either fetched from a URL you serve (**PDFURL**) or inlined into the page as base64 (**Base64Data**).

Rendering happens in the browser, on a canvas that follows the device pixel ratio, so the page stays sharp on high-density displays and is redrawn when the window is resized.

Quick Start

1. Drop a **TsgcHTMLComponent_PDFViewer** component on the form (or create it at runtime).

2. Set **PDFURL** to the URL your host serves the document from, and give the viewer a **Height**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_PDFViewer *oViewer = new TsgcHTMLComponent_PDFViewer(NULL);
try
{
  oViewer->PDFURL = "/docs/invoice.pdf";
  oViewer->Height = "800px";
  html = oViewer->HTML;
}
__finally
{
  delete oViewer;
}
```

Loading a Document

What it does: There are two ways to give the viewer a document, and you pick one.

PDFURL is a URL your host serves. The browser fetches the file itself, which keeps the page small and lets the browser cache the document. Use it for files that already live behind a route, and for documents big enough that in-lining them would bloat the page.

```
oViewer->PDFURL = "/docs/invoice.pdf";
```

Base64Data carries the PDF bytes inside the page itself, encoded as base64. Nothing extra is served and no route is needed, which suits documents you generate on the fly, documents that are short-lived, and documents you do not want to expose at a fetchable URL at all. The component accepts a plain base64 string or a full data URI: an optional `data:application/pdfbase64,` prefix is recognised and stripped.

```
oViewer->Base64Data = vBase64;   // "JVBERi0xLjQK..." or "data:application/pdfbase64,JVBERi0xLjQK..."
oViewer->DownloadFileName = "invoice-2026-07.pdf";
```

If both are set, the inlined bytes win: the viewer decodes **Base64Data** and never fetches **PDFURL**. If neither is set, the viewer shows *No PDF document has been specified.* in its error area. The same area reports a document that fails to load, base64 that fails to decode, and a page that fails to render.

Toolbar and Navigation

What it does: *ShowToolbar* is the master switch: set it to False and no toolbar is emitted at all, whatever the other switches say. With it True, each group is turned on or off on its own.

Property	Default	What appears in the toolbar
ShowPageNav	True	Previous page and next page buttons, plus a *Page n of m* indicator.
ShowZoomControls	True	Zoom out, zoom in and a fit button that returns to the base scale.
ShowSearch	True	A search box with previous match and next match buttons and a match counter.
ShowDownload	True	A download button that saves the document as DownloadFileName.
ShowPrint	True	A print button that sends the document to the browser's print dialog.

Pages: the viewer draws one page at a time and opens on *InitialPage*, clamped to the document (a page number below 1, or beyond the last page, is corrected). **Zoom** sets the base scale, and the zoom buttons then apply a factor of 1.2 on top of it, in either direction; *Fit* resets that factor to 1.

Search: typing a term and pressing Enter (or clicking a match button) scans the text of every page and collects the pages that contain the term, case-insensitively. The counter shows *match / total*, the viewer jumps to the first page that matches, and the previous / next buttons step through the matching pages. It is a page-level search: it takes you to the page, it does not highlight the word inside it.

Download and print: both work with either source. With **Base64Data** the decoded bytes are wrapped in a blob, so the file is saved or printed without a second request; with **PDFURL** the URL is used. Download names the file **DownloadFileName**; print loads the document into a hidden frame and calls the browser's print dialog, falling back to a new tab if the browser refuses.

Self-Hosting pdf.js

What it does: The viewer needs the pdf.js library and its worker. By default they are referenced from a public CDN:

Property	Default

PDFJSLibURL	https://cdn.jsdelivr.net/npm/pdfjs-dist@3.11.174/build/pdf.min.js
PDFJSWorkerURL	https://cdn.jsdelivr.net/npm/pdfjs-dist@3.11.174/build/pdf.worker.min.js

A CDN is convenient, but many deployments cannot use one: an air-gapped network, an intranet with no outbound access, a content security policy that allows scripts only from your own origin, or simply a policy of not depending on a third party. In all those cases, **self-host the two files**:

1. Download `pdf.min.js` and `pdf.worker.min.js` from the two URLs above (or from the pdfjs-dist package) and copy them next to your application.

2. Serve them from your own host, for example under `/js/`.

3. Point **PDFJSLibURL** and **PDFJSWorkerURL** at your own URLs. Nothing else changes.

```
oViewer->PDFJSLibURL = "/js/pdf.min.js";
oViewer->PDFJSWorkerURL = "/js/pdf.worker.min.js";
```

Keep the two files in step: the worker must come from the same pdf.js version as the library. The component uses the **UMD** build of pdf.js 3.x because it exposes the global `pdfjsLib` from a plain `<script src>` tag, with no ES module loader, which is how the sibling components embed Quill, Chart.js and Leaflet. If the library cannot be loaded, the viewer reports *The PDF viewer library could not be loaded.* in its error area instead of failing silently.

Security

Two values reach the emitted markup and script, and both are sanitised before they do.

The element id (**ViewerID**) is filtered down to `A-Z`, `a-z`, `0-9`, `_` and `-`. Everything else is discarded, so the id can never break out of an attribute nor out of a JavaScript string. If nothing survives the filter, or the property was left empty, the component uses the id it generates for itself on the first render, which is already inside that charset.

The **Base64Data** payload is reduced to the strict base64 alphabet (`A-Z`, `a-z`, `0-9`, `+`, `/`, `=`) plus the URL-safe `-` and `_`. Quotes, angle brackets, backslashes and every other character are dropped. A hostile payload therefore cannot become markup or script: at worst it decodes to nothing and the viewer reports that the data could not be decoded.

The URLs (**PDFURL**, **PDFJSLibURL**, **PDFJSWorkerURL**) are passed through the URL sanitiser and escaped for the attribute or the JavaScript string literal they land in.

Appearance

What it does: *Theme* picks the light or dark colour scheme, and *Width* and *Height* size the panel. The page canvas is centred in the scrolling view area and never exceeds the width of the panel.

Property	Type	Default	Description
PDFURL	string	''	URL the browser fetches the document from.
Base64Data	string	''	The PDF bytes inlined in the page as base64. Takes precedence over PDFURL.

ViewerID	string	''	The id of the generated root element. Sanitised, and the base of every child id. Empty means a unique id is generated on the first render.
Height	string	'600px'	Height of the viewer panel, as a CSS length.
Width	string	'100%'	Width of the viewer panel, as a CSS length.
InitialPage	Integer	1	Page shown when the document opens, clamped to the document.
Zoom	string	'auto'	Base scale: auto / page-width / fit, a factor such as 1.5, or a percentage such as 150%.
ShowToolbar	Boolean	True	Master switch for the toolbar.
ShowPage-Nav	Boolean	True	Previous / next page buttons and the page indicator.
ShowZoom-Controls	Boolean	True	Zoom out, zoom in and fit buttons.
ShowDown-load	Boolean	True	Download button.
ShowPrint	Boolean	True	Print button.
ShowSearch	Boolean	True	Search box, match buttons and match counter.
Theme	Ts-gcHTMLPDFView-erTheme	pvLight	Colour scheme: pvLight or pvDark.
PDFJSLibURL	string	jsDelivr pdf.min.js	URL of the pdf.js library. Point it at your own host to self-host.
PDFJS-WorkerURL	string	jsDelivr pdf.worker.min.js	URL of the pdf.js worker. Point it at your own host to self-host.
Download-FileName	string	'document.pdf'	File name suggested by the download button.

ViewerID is not hardcoded to a literal any more. Leave it empty and a unique element id is generated automatically the first time the component renders, and reused for the life of the instance, so two viewers on the same page never collide and pushed fragments keep targeting the element this instance actually rendered. That matters here because the id is what the inline script passes to `getElementById` to bind the canvas it draws the pages on: when two viewers shared one id, the script of the second one drove the canvas of the first. Set *ViewerID* explicitly if you need a stable, known id (for example to reach the element from your own JavaScript). The generated id is `sgcPDFViewer` plus a process-unique number, such as `sgcPDFViewer3`.

Output

The inherited **HTML** property returns a `<div>` carrying the sanitised *ViewerID*, the toolbar markup, an error area, a scrolling view holding the `<canvas>`, the `<script src>` reference to pdf.js and an inline initialization script that loads the document and wires up the toolbar. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 17 properties available on the component.
- Methods — this component publishes no methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_Modal

TsgcHTMLComponent_Modal generates a Bootstrap modal dialog (header, body and footer) together with an optional trigger button that opens it.

Introduction

TsgcHTMLComponent_Modal emits the markup of a Bootstrap `modal` dialog. You set the **Title**, **Body** and (optionally) a **Footer**, choose a **Size** and whether the dialog is vertically centered, scrollable or uses a static backdrop, then read the inherited **HTML** property to get the complete markup. The modal stays hidden until a button or link with `data-bs-toggle="modal"` targets its **ModalID**; use **BuildTriggerButton** to generate that button.

The component relies on the Bootstrap JavaScript bundle, which the sgcHTML engine serves from an embedded resource, so no Content Delivery Network is required at runtime.

Quick Start

1. Drop a **TsgcHTMLComponent_Modal** component on the form (or create it at runtime).

2. Set the **ModalID**, **Title** and **Body**, then add footer buttons with **AddFooterButton**.

3. Read the **HTML** property and serve it, and emit a trigger button with **BuildTriggerButton**.

```
TsgcHTMLComponent_Modal *oModal = new TsgcHTMLComponent_Modal(NULL);
try
{
  oModal->ModalID = "confirmModal";
  oModal->Title = "Delete Record";
  oModal->Body = "Are you sure you want to delete this record?";
  oModal->Size = msDefault;
  oModal->AddFooterButton("Cancel", bsSecondary, true);
  oModal->AddFooterButton("Delete", bsDanger, true);
  html = TsgcHTMLComponent_Modal::BuildTriggerButton("confirmModal", "Delete", bsDanger)
    + oModal->HTML;
}
__finally
{
  delete oModal;
}
```

Content and Layout

What it does: *Title*, *Body* and *Footer* hold the three regions of the dialog. *ModalID* is the element id a trigger button targets. *Size* selects one of the Bootstrap modal widths. *Centered*, *Scrollable* and *StaticBackdrop* change how the dialog behaves, and *ShowClose* toggles the header close (×) button. *CSSClass* adds extra classes for custom styling.

Property	Type	Default	Description

Title	string	''	Heading drawn in the modal header. When empty the header is omitted. Plain text: markup in it is shown literally.
Body	string	''	HTML content of the modal body.
Footer	string	''	Raw HTML for the footer. Overrides any buttons added with AddFooterButton.
ModalID	string	'sgcModal'	The id of the generated modal element. A trigger button targets this id.
Size	TsgcHTMLModalSize	msDefault	Dialog width: msDefault, msSmall, msLarge, msXLarge or msFullscreen.
Centered	Boolean	True	Vertically centers the dialog in the viewport.
Scrollable	Boolean	False	Makes the modal body scroll when its content is taller than the viewport.
StaticBackdrop	Boolean	False	Keeps the dialog open when the backdrop is clicked or Esc is pressed.
ShowClose	Boolean	True	Shows the header close (×) button.
CSSClass	string	''	Extra CSS classes appended for custom styling.

Footer Buttons

What it does: Instead of supplying raw footer HTML through *Footer*, call **AddFooterButton** once per button. Each call appends a styled `<button>` to the footer and can optionally dismiss the modal when clicked. If you set neither *Footer* nor any footer button, the component draws a single localized Close button.

```
oModal->AddFooterButton("Cancel", bsSecondary, true);
oModal->AddFooterButton("Save", bsPrimary);
```

One-Line Builders

What it does: The static **Build** method returns a complete modal in a single call, without creating an instance. **BuildTriggerButton** returns the matching button markup that opens a modal by its id. Both are `class function` helpers you can call on the type itself.

```
String sButton = TsgcHTMLComponent_Modal::BuildTriggerButton("infoModal", "More Info", bsPrimary);
String sModal  = TsgcHTMLComponent_Modal::Build("infoModal", "Details", "Full description here.", "", msLarge
```

Output

The inherited **HTML** property returns the full `<div class="modal fade">` markup including the dialog, content, header, body and footer. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document. Place the trigger button markup wherever the user should open the dialog from.

Reference

- Properties — 10 properties available on the component.
- Methods — 3 public methods (1 instance, 2 static builders).
- Events — this component has no published events.

TsgcHTMLComponent_Offcanvas

TsgcHTMLComponent_Offcanvas generates a Bootstrap offcanvas panel that slides in from a screen edge, together with an optional trigger button that opens it.

Introduction

TsgcHTMLComponent_Offcanvas emits the markup of a Bootstrap `offcanvas` panel: a sidebar that slides in from the start, end, top or bottom edge of the viewport. You set the **Title** and **Body**, choose a **Placement** and whether the panel uses a dark theme or a static backdrop, then read the inherited **HTML** property to get the complete markup. The panel stays hidden until a button with `data-bs-toggle="offcanvas"` targets its **OffcanvasID**; use **BuildTriggerButton** to generate that button.

The component relies on the Bootstrap JavaScript bundle, which the sgcHTML engine serves from an embedded resource, so no Content Delivery Network is required at runtime.

Quick Start

1. Drop a **TsgcHTMLComponent_Offcanvas** component on the form (or create it at runtime).

2. Set the **OffcanvasID**, **Title**, **Body** and a **Placement**.

3. Read the **HTML** property and serve it, and emit a trigger button with **BuildTriggerButton**.

```
TsgcHTMLComponent_Offcanvas *oOC = new TsgcHTMLComponent_Offcanvas(NULL);
try
{
  oOC->OffcanvasID = "menuPanel";
  oOC->Title = "Navigation";
  oOC->Body = "<ul><li>Home</li><li>Reports</li></ul>";
  oOC->Placement = opStart;
  html = TsgcHTMLComponent_Offcanvas::BuildTriggerButton("menuPanel", "Menu", bsPrimary)
    + oOC->HTML;
}
__finally
{
  delete oOC;
}
```

Content and Placement

What it does: *Title* and *Body* hold the heading and content of the panel. *OffcanvasID* is the element id a trigger button targets. *Placement* chooses the edge the panel slides in from. *Dark* applies the dark theme, *StaticBackdrop* keeps the panel open when the backdrop is clicked, and *ShowClose* toggles the header close (×) button. *CSSClass* adds extra classes for custom styling.

Property	Type	Default	Description

Title	string	''	Heading drawn in the offcanvas header. Plain text: markup in it is shown literally.
Body	string	''	HTML content of the offcanvas body.
Offcan-vasID	string	'sgcOff-canvas'	The id of the generated panel. A trigger button targets this id.
Placement	TsgcHTMLOff-canvasPlace-ment	opStart	Edge the panel slides in from: opStart, opEnd, opTop or opBottom.
Dark	Boolean	False	Applies the dark theme (text-bg-dark and a white close button).
StaticBack-drop	Boolean	False	Keeps the panel open when the backdrop is clicked.
ShowClose	Boolean	True	Shows the header close (×) button.
CSSClass	string	''	Extra CSS classes appended for custom styling.

One-Line Builders

What it does: The static **Build** method returns a complete offcanvas panel in a single call, without creating an instance. **BuildTriggerButton** returns the matching button markup that opens a panel by its id. Both are `class function` helpers you call on the type itself.

```
String sButton = TsgcHTMLComponent_Offcanvas::BuildTriggerButton("cartPanel", "Cart", bsSuccess);
String sPanel  = TsgcHTMLComponent_Offcanvas::Build("cartPanel", "Your Cart", "<p>Empty</p>", opEnd);
```

Output

The inherited **HTML** property returns the full `<div class="offcanvas-...">` markup including the header, optional close button and body. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document. Place the trigger button markup wherever the user should open the panel from.

Reference

- Properties — 8 properties available on the component.
- Methods — 2 static builder methods.
- Events — this component has no published events.

TsgcHTMLComponent_Popover

TsgcHTMLComponent_Popover wraps any trigger element (or a generated button) with a Bootstrap popover that shows a title and body when clicked, hovered or focused.

Introduction

TsgcHTMLComponent_Popover emits a trigger element decorated with the Bootstrap `data-bs-toggle="popover"` attributes plus, optionally, the small script that initializes every popover on the page. You supply the trigger markup through **Content** (or let the component build a button from **Title** and **ContentStyle**), set the popover **Title** and **Body**, then choose how it is shown with **Placement** and **Trigger**. Read the inherited **HTML** property to get the complete markup.

The component relies on the Bootstrap JavaScript bundle, which the sgcHTML engine serves from an embedded resource, so no Content Delivery Network is required at runtime.

Quick Start

1. Drop a **TsgcHTMLComponent_Popover** component on the form (or create it at runtime).

2. Set the **Title** and **Body**, and either assign **Content** or pick a **ContentStyle** to generate a button.

3. Read the **HTML** property and serve it; leave **AutoInit** True so the popover initializes itself.

```
TsgcHTMLComponent_Popover *oPop = new TsgcHTMLComponent_Popover(NULL);
try
{
  oPop->Title = "More Info";
  oPop->Body = "This field accepts a valid email address.";
  oPop->Placement = plTop;
  oPop->Trigger = ptHover;
  oPop->ContentStyle = bsInfo;
  html = oPop->HTML;
}
__finally
{
  delete oPop;
}
```

Trigger and Content

What it does: *Content* holds the markup of the element the popover hangs off; when empty the component builds a button from *Title* using *ContentStyle*. *Title* and *Body* are the popover heading and text. *PopoverID* sets the element id. *Dismissible* switches to focus-trigger so the popover dismisses on the next outside click.

Property	Type	Default	Description
Content	string	''	Markup of the trigger element. When empty a button is generated from Title and ContentStyle.

Title	string	''	The popover heading, also used as the generated button caption.
Body	string	''	The popover content text (data-bs-content). Plain text: markup in it is shown literally by the popover.
ContentStyle	TsgcHTMLButtonStyle	bsPrimary	Button style used when Content is empty and a button is generated.
PopoverID	string	''	Optional id for the trigger element.
Dismissible	Boolean	False	Dismisses the popover on the next click outside it (focus trigger).

Placement and Initialization

What it does: *Placement* chooses which side the popover appears on and *Trigger* chooses what activates it. *AutoInit* appends a small initialization script so the popover works without extra wiring, and *InitScript* holds that script so you can replace it.

Property	Type	Default	Description
Placement	TsgcHTMLPlacement	plTop	Side the popover appears on: plTop, plBottom, plLeft or plRight.
Trigger	TsgcHTMLPopoverTrigger	ptClick	What activates the popover: ptClick, ptHover, ptFocus or ptManual.
AutoInit	Boolean	True	Appends the initialization script so popovers activate automatically.
InitScript	string	(see remarks)	The JavaScript appended when AutoInit is True.

```
String sHtml = TsgcHTMLComponent_Popover::BuildButton("Help", "Usage", "Click to learn more.", bsSecondary, p
```

Output

The inherited **HTML** property returns a `<span>` wrapping the trigger element, carrying the popover attributes, followed by the initialization `<script>` when **AutoInit** is True. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 10 properties available on the component.
- Methods — 2 static builder methods.
- Events — this component has no published events.

TsgcHTMLComponent_Toast

TsgcHTMLComponent_Toast generates a Bootstrap toast: a small, non-blocking notification with a title, body, timestamp and optional auto-hide, ready to drop into a positioned container.

Introduction

TsgcHTMLComponent_Toast emits the markup of a Bootstrap `toast` notification. You set the **Title** and **Body**, pick a contextual color, optionally add a **Timestamp** and an **Icon**, and decide whether it hides itself after a **Delay**. Read the inherited **HTML** property to get the toast markup, then wrap one or more toasts in a positioned container with the static **BuildContainer** helper.

The component relies on the Bootstrap JavaScript bundle, which the sgcHTML engine serves from an embedded resource, so no Content Delivery Network is required at runtime.

Quick Start

1. Drop a **TsgcHTMLComponent_Toast** component on the form (or create it at runtime).

2. Set the **Title**, **Body** and a color, then read the **HTML**.

3. Wrap the toast markup in a positioned container with **BuildContainer**.

```
TsgcHTMLComponent_Toast *oToast = new TsgcHTMLComponent_Toast(NULL);
try
{
  oToast->Title = "Saved";
  oToast->Body = "Your changes have been saved.";
  oToast->ColorStyle = hcSuccess;
  oToast->Timestamp = "just now";
  html = TsgcHTMLComponent_Toast::BuildContainer(oToast->HTML, tpTopEnd);
}
__finally
{
  delete oToast;
}
```

Content and Appearance

What it does: *Title* and *Body* hold the heading and message. *Color* takes a raw Bootstrap color name, or use *ColorStyle* to pick one from an enum. *Timestamp* and *Icon* add an optional time and a leading icon to the header. *ToastID* sets the element id.

Property	Type	Default	Description
Title	string	''	Heading drawn in the toast header. Plain text: markup in it is shown literally.
Body	string	''	The toast message.

Time-stamp	string	''	Optional time text shown to the right of the header. Plain text.
Color	string	''	Raw Bootstrap color name (for example success). Overrides Col-orStyle when set.
ColorStyle	Ts-gcHTML-Color	hcPri-mary	Contextual color picked from an enum, used when Color is emp-ty.
ToastID	string	''	Optional id for the toast element.
Icon	string	''	Optional icon markup shown before the title.

Auto-Hide

What it does: *AutoHide* decides whether the toast dismisses itself, and *Delay* sets how long it stays visible first. With AutoHide off the toast stays until the user closes it.

Property	Type	Default	Description
AutoHide	Boolean	True	When True the toast hides itself after Delay milliseconds.
Delay	Integer	5000	Milliseconds the toast stays visible before auto-hiding.

```
String sToast = TsgcHTMLComponent_Toast::Build("Error", "Could not connect.", hcDanger, "");
String sHtml  = TsgcHTMLComponent_Toast::BuildContainer(sToast, tpBottomEnd);
```

Output

The inherited **HTML** property returns the `<div class="toast show">` markup including the header (icon, title, time-stamp and close button) and the body. Several toasts share a single container, so wrap your toast markup with **BuildContainer** to position them, then serve the result or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap**.

Reference

- Properties — 9 properties available on the component.
- Methods — 2 static builder methods.
- Events — this component has no published events.

TsgcHTMLComponent_Snackbar

TsgcHTMLComponent_Snackbar renders a brief, self-dismissing toast message anchored to a corner or edge of the viewport, with an optional action link or button.

Introduction

TsgcHTMLComponent_Snackbar emits a fixed-position `<div>` that floats above the page content to show a short message, such as the result of a background action. It is styled by a small block of CSS the component injects, so it does not depend on any external toast library. The complete markup is returned through the inherited **HTML** property.

You set the **Message**, pick a **Color** and a **Position**, and optionally add an **ActionText**. When **AutoHide** is True (the default) the component also emits a tiny inline script that removes the snackbar after **Delay** milliseconds.

Quick Start

1. Drop a **TsgcHTMLComponent_Snackbar** component on the form (or create it at runtime).

2. Set the **Message**, then choose a **Color** and a **Position**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Snackbar *oSnackbar = new TsgcHTMLComponent_Snackbar(NULL);
try
{
  oSnackbar->Message = "Settings saved";
  oSnackbar->Color = hcSuccess;
  oSnackbar->Position = sbBottomRight;
  oSnackbar->ActionText = "Undo";
  html = oSnackbar->HTML;
}
__finally
{
  delete oSnackbar;
}
```

Message and Action

What it does: *Message* is the text shown in the snackbar. *ActionText* adds an inline action; when *ActionHref* is set the action is a link, otherwise it is a button that simply dismisses the snackbar. *SnackbarID* sets the id of the generated element, which the auto-hide script uses to find and remove it.

Property	Type	Default	Description
Message	string	''	The text displayed in the snackbar.
ActionText	string	''	Caption of the optional action. Empty means no action.

ActionHref	string	''	If set, the action is a link to this URL; otherwise the action is a dismiss button.
SnackbarID	string	'sgcSnackbar'	The id of the generated element.

Appearance and Behavior

What it does: *Color* selects the Bootstrap background color, *Position* anchors the snackbar to an edge or corner of the viewport, and *AutoHide* together with *Delay* controls whether and when the snackbar removes itself.

Property	Type	Default	Description
Color	TsgcHTMLColor	hcDark	Background color: hcPrimary, hcSecondary, hcSuccess, hcDanger, hcWarning, hcInfo, hcLight, hcDark, hcWhite or hcMuted.
Position	TsgcHTMLSnackbarPosition	sbBottom	Anchor point: sbBottom, sbTop, sbBottomLeft, sbBottomRight, sbTopLeft or sbTopRight.
AutoHide	Boolean	True	When True the snackbar removes itself after Delay milliseconds.
Delay	Integer	4000	Milliseconds to keep the snackbar visible when AutoHide is True.

```
oSnackbar->Color = hcDanger;
oSnackbar->Position = sbTop;
oSnackbar->AutoHide = true;
oSnackbar->Delay = 6000;
```

One-call Build

What it does: When you do not need an instance, the static **Build** class function creates a snackbar, applies a message, color, action text and position, and returns the markup in a single call.

```
html = TsgcHTMLComponent_Snackbar::Build("Profile updated", hcSuccess, "View", sbBottomRight);
```

Output

The inherited **HTML** property returns a fixed-position `<div>` holding the message and the optional action, followed by the auto-hide `<script>` when **AutoHide** is True. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 8 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Notification

TsgcHTMLComponent_Notification renders a bell button with an unread-count badge that opens a dropdown list of notification items, and updates a single notification and the badge in the browser by pushing small fragments over the WebSocket connection.

Introduction

TsgcHTMLComponent_Notification emits a Bootstrap dropdown: a bell button shows an unread-count badge, and clicking it reveals a list of notifications. Each notification has a title, a message, a timestamp, a color dot and an optional link. The component builds the whole structure for you and returns it through the inherited **HTML** property.

Notifications live in the **Items** collection of **TsgcHTMLNotificationItem** objects. You can fill the collection at design time, add items in code, or use the **AddNotification** helper. The number of unread items is available through the read-only **UnreadCount** property (and through **GetUnreadCount**), and the badge markup alone can be obtained with **GetBadgeHTML**.

Notifications arrive while the page is open, so the component does not stop there. Give a notification an **Id** and it gets a stable element id in the page, which means that notification can be rendered on its own and sent to the browsers that are already connected, with no reload and no polling. See **Live Updates over WebSockets** below.

Quick Start

1. Drop a **TsgcHTMLComponent_Notification** component on the form (or create it at runtime).

2. Add notifications with **AddNotification** (or fill the **Items** collection).

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Notification *oNotification = new TsgcHTMLComponent_Notification(NULL);
try
{
  oNotification->Title = "Notifications";
  oNotification->AddNotification("Order shipped", "Order #4711 is on its way", hcSuccess);
  oNotification->AddNotification("New message", "You have a reply from support", hcInfo);
  html = oNotification->HTML;
}
__finally
{
  delete oNotification;
}
```

Later, when a notification arrives or is read, do not rebuild the page. Change that one notification and push it, as shown in **Live Updates over WebSockets**.

Notifications

What it does: *Items* is the collection of notifications drawn in the dropdown. Each **TsgcHTMLNotificationItem** exposes *Id*, *Title*, *Message*, *Timestamp*, *Read*, *Color*, *Icon* and *Href*. Unread items are highlighted and counted. *MaxVisible* caps how many items are shown, and *EmptyText* is the message displayed when there are none.

Property	Type	Default	Description
Items	TsgcHTMLNotifica-tionItems	empty	Collection of TsgcHTMLNotificationItem objects drawn in the dropdown.
MaxVisi-ble	Integer	5	Maximum number of items shown before a “View all” link appears.
Empty-Text	string	‘No new notifica-tions’	Message shown when the collection is empty.

```
TsgcHTMLNotificationItem *item = oNotification->Items->Add();
item->Id = "bk1";
item->Title = "Backup complete";
item->Message = "Nightly backup finished successfully";
item->Color = hcSuccess;
item->Read = false;
```

Titles, messages and timestamps are escaped when they are written, and an item's *Href* is sanitized, so only safe schemes survive. Values that come from a database or from a user cannot inject markup into the page.

Bell and Badge

What it does: *Title* is the heading shown at the top of the dropdown. *BellIcon* is the glyph on the trigger button. *ShowBadge* toggles the unread-count badge, and the read-only *UnreadCount* reports how many items are unread. *NotificationID* sets the id of the dropdown container, and it is also the prefix of the element id of every notification row and of the badge.

Property	Type	Default	Description
Title	string	‘Notifica-tions’	Heading shown at the top of the dropdown.
BellIcon	string	‘🔔’	Glyph or HTML entity used for the bell trigger button.
Show-Badge	Boolean	True	Shows the unread-count badge on the bell button. The badge always carries a stable element id, and it is hidden while noth-ing is unread.
Notifica-tionID	string	‘sgcNotifica-tions’	The id of the generated dropdown element, and the pre-fix of every notification element id.

Live Updates over WebSockets

What it does: a notification arrives, or the user reads one, while the page is already open. Instead of re-rendering the page and instead of polling, you change that one notification on the server and push it to the browsers that are already connected.

Member	Description
Id (item property)	Your key for the notification. The stable element id of the item's row is derived from it, so only items that carry an Id can be pushed.
`function AddNotification(const aId, aTitle, aMessage: string; aColor: TsgcHTMLColor = hcPrimary; const aTimestamp: string = ''): TsgcHTMLNotificationItem;`	Adds the notification with that id and returns the item, so you can still fill Icon, Href or Read. Calling it twice with the same id updates the item instead of duplicating it.
`procedure MarkAsRead(const aId: string);`	Flips the notification with that id to read: the row loses its highlight and the unread count goes down by one. An unknown id does nothing.
`function GetUnreadCount: Integer;`	The number of items whose Read is False. This is the number the badge shows, and it also backs the read-only UnreadCount property.
`function GetNotificationFragmentHTML(const aId: string): string;`	The markup of that one notification, marked as an out-of-band swap (`hx-swap-oob`) on the element id of its row. When no item carries that id, a delete fragment for the same element id is returned instead.
`function GetBadgeFragmentHTML: string;`	The refreshed unread-count badge, marked as an out-of-band swap on the badge element id. Empty when ShowBadge is False.

The fragment is sent to the browser through the WebSocket connection the page already holds, using the sgcHTML engine, TsgcHTMX_Engine_Server:

Engine Method	Description
`procedure PushFragment(const aGuid, aHTML: string);`	Sends the fragment to a single client, addressed by its connection guid.
`procedure BroadcastFragment(const aHTML: string; const aChannel: string = ''; const aExclude: string = '');`	Sends the fragment to every connected client, optionally only those on a channel, optionally excluding one guid.

The browser applies the swap in place: the element carrying that id is replaced, and the rest of the page is left alone. There is no HTTP round-trip and no client-side data binding, the server owns the list and pushes only the difference.

The flow is always the same, whatever changed: **change the notification, then push it, then push the badge**. The badge is a separate element with its own id, so pushing **GetBadgeFragmentHTML** whenever the unread count moved keeps the number on the bell in step with the list.

```
// a notification arrived: add it with an id, then push the row and the badge
void __fastcall TServerForm::NotifyOrderShipped(const String aId, const String aOrder)
{
  TsgcHTMLNotificationItem *oItem = oNotification->AddNotification(aId, "Order shipped",
    "Order " + aOrder + " is on its way", hcSuccess);
  oItem->Href = "/orders/" + aOrder;
  oEngine->BroadcastFragment(oNotification->GetNotificationFragmentHTML(aId));
  oEngine->BroadcastFragment(oNotification->GetBadgeFragmentHTML());
}

// the user read it: the SAME two pushes refresh the row and the badge
void __fastcall TServerForm::NotificationRead(const String aId)
{
  oNotification->MarkAsRead(aId);
  oEngine->BroadcastFragment(oNotification->GetNotificationFragmentHTML(aId));
  oEngine->BroadcastFragment(oNotification->GetBadgeFragmentHTML());
}
```

Use **PushFragment** instead of **BroadcastFragment** when the notification belongs to one user and only that browser should see it, and pass a channel to **BroadcastFragment** when the list is per team or per room rather than global.

Output

The inherited **HTML** property returns a Bootstrap dropdown: a bell `<button>` carrying the badge, and a `dropdown-menu` holding the title and the scrollable list of items. Each item that has an **Id** is written with its own stable element id, and the badge always has one, so both can be replaced later by an out-of-band swap. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 8 properties available on the component.
- Methods — 6 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_NotificationInbox

A bell with an unread badge and the notifications of one user, with the host owning every row.

Introduction

TsgcHTMLComponent_NotificationInbox is a bell with an unread badge and a dropdown list of the notifications of ONE user: a title, a text, a time, a read state and an optional link.

The library **stores nothing**. The host answers OnLoadNotifications with the rows it keeps, and is told through OnMarkRead and OnMarkAllRead what the human did, so it can persist it. The component only holds what it is rendering.

It renders UI and **enforces no authorization**. Every action it emits travels over the channel as data sent by the client, so no user id is ever written into the markup: the host resolves the acting user from the request session and checks that the user owns the notification id that arrived.

It lives in the unit **sgcHTML_Component_NotificationInbox** and works on every edition, over WebSocket or SSE. Delivering to a browser that is not on the page is what needs TsgcHTMLWebPush, which is Enterprise and All-Access.

Quick Start

1. Create the component, give it an **InboxID** and handle the three events.

2. Route the three actions it posts (inboxMarkRead, inboxMarkAllRead, inboxRefresh) into **ProcessAction**, with the user taken from the session.

3. Push the affected markup back with the fragment methods instead of re-rendering the page.

The Actions It Posts

Action	Fields	What the host calls
inboxMarkRead	action, inbox, id	MarkRead(user, id)
inboxMarkAllRead	action, inbox	MarkAllRead(user)
inboxRefresh	action, inbox	LoadNotifications(user)

The inbox field carries the element id of the inbox that rendered the form, which is how a page with several inboxes routes the action to the right component. No form carries a user.

TsgcHTMLComponent_NotificationPreferences

One row per category, one checkbox per channel, and the host owning every value.

Introduction

TsgcHTMLComponent_NotificationPreferences is the table a human ticks the delivery channels in: one row per category, one checkbox per channel (in app, push, email).

The library **stores nothing**. OnLoadPreferences asks the host for the values of one user, OnSavePreferences hands back the values a human ticked, and persisting them is the host's job. The values are name=value pairs, one per offered category and channel:

```
orders.inapp=1
orders.push=0
orders.email=1
```

A value is 1 when the channel is on and 0 when it is off, and **every offered pair is always present** in what the save event receives, so the host never has to guess what an absent key meant.

The saved form travels over the channel as data sent by the client, so no user id is ever written into the markup, and values that name a category or a channel this component does not offer are dropped. A forged form cannot widen the preference set.

It lives in the unit **sgcHTML_Component_NotificationPreferences**. The channel names are shared with TsgcHTMLComponent_Notification, which asks the host for them in OnGetChannel before it delivers anything.

Quick Start

TsgcHTMLWebPush

Web Push with VAPID for sgcHTML: it validates subscriptions, hands them to the host and sends notifications.

Introduction

TsgcHTMLWebPush is Web Push (RFC 8030) with VAPID (RFC 8292) for sgcHTML: it validates what a browser posts when it subscribes, hands every subscription to the application as plain strings, and sends a notification to the subscriptions of one user, encrypted with the aes128gcm scheme of RFC 8291.

Editions. Web Push is available in the **Enterprise** and **All-Access** editions. The unit is compiled in only when SGC_WEBPUSH is defined, which sgcVer.inc sets for those two editions. The notification inbox and the preferences table are part of sgcHTML itself and work on every edition over WebSocket or SSE: only the push channel, which reaches a browser that is not on the page, needs the define.

The component **keeps no subscription of its own**. Every subscription is handed to the host through OnSaveSubscription and OnDeleteSubscription and read back through OnLoadSubscriptions, so a table, a file or a cache all work and nothing is lost when the process restarts.

It lives in the unit **sgcHTML_WebPush**, and the page side is written by TsgcHTMLTemplate_Bootstrap once WebPushEnabled is on. See Web Push for how the pieces fit together.

Quick Start

1. Generate a VAPID pair once with GenerateVAPIDKeys and store it.

2. Assign the pair and a Subject, handle the three storage events and set Enabled.

3. Assign the component to the WebPush property of TsgcHTMX_Engine_Server and set Template.WebPushEnabled to True. The engine then serves the two endpoints and the service worker, and the page carries the subscribe script.

4. Call Send when something happens the user should know about.

```
FPush = new TsgcHTMLWebPush(this);
FPush->VAPIDPublicKey = ReadSetting("vapid.public");
FPush->VAPIDPrivateKey = ReadSetting("vapid.private");
FPush->Subject = "mailto:support@example.com";
FPush->Enabled = true;

oHTMX->WebPush = FPush;
oHTMX->Template->WebPushEnabled = true;

FPush->Send("u1", "Order shipped", "Order 1042 left the warehouse", "/orders/1042");
```

What Is Validated

Everything a browser posts is treated as hostile text. A subscription is stored only when the endpoint is an https URL of at most 2048 printable US-ASCII characters with a host, p256dh is base64url of exactly 65 bytes beginning with 0x04, and auth is base64url of exactly 16 bytes. A body that is not JSON, or that carries anything else, is a refusal and never an exception.

The user is never read from the posted body. It comes from the session of a TsgcHTMLAuth, or from OnResolveUser, and with neither of them the engine refuses the request.

TsgcHTMLComponent_Spinner

TsgcHTMLComponent_Spinner renders a Bootstrap loading spinner (border or grow), in a chosen color and size, with hidden text for assistive technology.

Introduction

TsgcHTMLComponent_Spinner emits a Bootstrap `spinner-border` or `spinner-grow` element you can drop into a page to show that work is in progress. The component builds the right class list for the chosen type, size and color, and adds a visually hidden label for screen readers. The complete markup is returned through the inherited **HTML** property.

Bootstrap supplies the spinner animation, so the emitted markup is a single small `<div>` with no extra script.

Quick Start

1. Drop a **TsgcHTMLComponent_Spinner** component on the form (or create it at runtime).

2. Choose a **SpinnerType**, a **Color** and a **Size**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Spinner *oSpinner = new TsgcHTMLComponent_Spinner(NULL);
try
{
  oSpinner->SpinnerType = spBorder;
  oSpinner->Color = "primary";
  oSpinner->Size = ssNormal;
  html = oSpinner->HTML;
}
__finally
{
  delete oSpinner;
}
```

Type and Size

What it does: *SpinnerType* chooses between the rotating border spinner and the pulsing grow spinner. *Size* switches between the normal size and a smaller variant. *SpinnerID* sets the id of the generated element.

Property	Type	Default	Description
SpinnerType	TsgcHTMLSpinnerType	spBorder	Spinner style: spBorder (rotating ring) or spGrow (pulsing dot).
Size	TsgcHTMLSpinnerSize	ssNormal	Spinner size: ssNormal or ssSmall.

SpinnerID	string	”	Optional id of the generated element.

Color and Label

What it does: *Color* sets the spinner color as a Bootstrap color name string, and *ColorStyle* offers the same choice as a **TsgcHTMLColor** enum. When *Color* is not empty it wins; clear it to let *ColorStyle* take effect. *Text* is the visually hidden label read by assistive technology.

Property	Type	Default	Description
Color	string	‘primary’	Bootstrap color name (primary, success, danger, ...). Takes precedence over ColorStyle when set.
ColorStyle	TsgcHTMLColor	hcPrimary	Color as an enum, used only when Color is empty.
Text	string	locale “Loading...”	Visually hidden label for screen readers.

```
oSpinner->SpinnerType = spGrow;
oSpinner->Color = "success";
oSpinner->Size = ssSmall;
oSpinner->Text = "Loading...";
```

One-call Build

What it does: The static **Build** class function returns the spinner markup in a single call. It is overloaded: pass the color as a Bootstrap name string, or as a **TsgcHTMLColor** enum.

```
html = TsgcHTMLComponent_Spinner::Build(spBorder, "primary", ssSmall);
// or with the enum overload
html = TsgcHTMLComponent_Spinner::Build(spGrow, hcSuccess);
```

Output

The inherited **HTML** property returns a single `<div>` carrying the spinner classes and a `role="status"` attribute, wrapping a visually hidden `<span>` label. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 6 properties available on the component.
- Methods — 1 public method (two overloads).
- Events — this component has no published events.

TsgcHTMLComponent_Placeholder

TsgcHTMLComponent_Placeholder renders a Bootstrap skeleton placeholder (a card or block of greyed-out lines, with optional image, title and buttons) shown while real content is loading.

Introduction

TsgcHTMLComponent_Placeholder emits a Bootstrap card filled with placeholder bars that animate while the page waits for data. It is a "skeleton screen": you show the placeholder first, then swap it for the real markup once the content arrives. The complete markup is returned through the inherited **HTML** property.

You control how many lines are drawn, their size and color, the animation style, and whether an image area, a title bar and a pair of placeholder buttons are included. Bootstrap supplies the placeholder styling and the glow or wave animation.

Quick Start

1. Drop a **TsgcHTMLComponent_Placeholder** component on the form (or create it at runtime).

2. Set the **LineCount**, choose an **Animation**, and turn on **ShowImage**, **ShowTitle** or **ShowButtons** as needed.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Placeholder *oPlaceholder = new TsgcHTMLComponent_Placeholder(NULL);
try
{
  oPlaceholder->LineCount = 3;
  oPlaceholder->Animation = paGlow;
  oPlaceholder->ShowImage = true;
  oPlaceholder->ShowButtons = true;
  html = oPlaceholder->HTML;
}
__finally
{
  delete oPlaceholder;
}
```

Layout

What it does: *LineCount* sets how many placeholder text lines are drawn. *ShowImage*, *ShowTitle* and *ShowButtons* add an image area, a title bar and a pair of placeholder buttons. *PlaceholderID* sets the id of the outer card.

Property	Type	Default	Description
LineCount	Integer	3	Number of placeholder text lines drawn in the card body.
ShowImage	Boolean	False	Adds a colored image area at the top of the card.
ShowTitle	Boolean	True	Adds a placeholder title bar above the text lines.

ShowButtons	Boolean	False	Adds two placeholder buttons below the text lines.
PlaceholderID	string	''	Optional id of the outer card element.

Size, Animation and Color

What it does: *Size* sets the height of the placeholder bars, *Animation* chooses the glow or wave shimmer, and *Color* tints the placeholder bars and image area.

Property	Type	Default	Description
Size	TsgcHTMLPlaceholder-Size	phNormal	Bar height: phSmall, phNormal, phLarge or phExtraLarge.
Anima-tion	TsgcHTMLPlacehold-erAnimation	paGlow	Shimmer style: paGlow or paWave.
Color	TsgcHTMLColor	hcSec-ondary	Tint of the placeholder bars and image area.

```
oPlaceholder->Size = phLarge;
oPlaceholder->Animation = paWave;
oPlaceholder->Color = hcSecondary;
```

One-call Build

What it does: Three static class functions return ready-made placeholders: **BuildCard** for a full card with image, title, lines and buttons, **BuildText** for just a block of text lines, and **BuildButton** for a single placeholder button.

```
html = TsgcHTMLComponent_Placeholder::BuildCard(3, paGlow);
html = TsgcHTMLComponent_Placeholder::BuildText(4, paWave);
html = TsgcHTMLComponent_Placeholder::BuildButton(paGlow);
```

Output

The inherited **HTML** property returns a Bootstrap `card` `<div>` containing the optional image area and a card body of placeholder bars wrapped in the chosen animation class. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 8 properties available on the component.
- Methods — 3 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_EmptyState

TsgcHTMLComponent_EmptyState renders the "nothing here yet" placeholder panel that a grid, a list or a search result shows when it has no rows to display, and it emits no script at all.

Introduction

TsgcHTMLComponent_EmptyState renders an optional visual (an icon or an image), a title, a description and up to two call-to-action buttons, wrapped in a single `<div class="sgc-empty-state">`. Use it wherever a collection component has zero rows to show: an empty search result, a filtered grid with no matches, a list before its first item is added.

No script, ever. Unlike almost every other interactive sgcHTML component, TsgcHTMLComponent_EmptyState emits no JavaScript at all, not even an inline snippet. That makes it safe to push as an HTMX out-of-band fragment as many times as the page needs: a grid that empties and refills over and over can swap the same empty-state markup back in on every response with zero risk of accumulating duplicate script tags or event listeners, because there is never a script tag to accumulate in the first place.

Quick Start

1. Drop a **TsgcHTMLComponent_EmptyState** component on the form (or create it at runtime).

2. Set **Icon**, **Title**, **Description**, **ActionCaption** and **ActionHref** as needed.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent, or push it as an HTMX out-of-band swap).

```
TsgcHTMLComponent_EmptyState *oEmptyState = new TsgcHTMLComponent_EmptyState(NULL);
try
{
  oEmptyState->Icon = "bi bi-inbox";
  oEmptyState->Title = "No results found";
  oEmptyState->Description = "Try a different search or clear your filters.";
  oEmptyState->ActionCaption = "Clear filters";
  oEmptyState->ActionHref = "/search?reset=1";
  html = oEmptyState->HTML;
}
__finally
{
  delete oEmptyState;
}
```

One-line shortcut. When all you need is a title and a description, the class function **Build** constructs a temporary instance, sets both properties, renders it and frees the instance in a single call, with no component for you to create or hold on to:

```
UnicodeString html = TsgcHTMLComponent_EmptyState::Build("No results found", "Try a different search term.");
```

Reach for **Build** for the plain "nothing to show" case; reach for the full property-by-property path above the moment you need an icon, an image or a call-to-action button, none of which Build has a parameter for.

Text

What it does: *Title* is the panel's headline; *Description* is the supporting copy shown under it. Both are HTML-encoded when rendered, so they are the safe pair to bind directly to database or user-supplied text, for example echoing a search query back to the visitor ("No results for 'kitten sweater'"). Contrast this with **Icon** below, which is emitted verbatim and must never carry end-user input.

Property	Type	Default	Description
Title	string	''	Headline text of the panel, HTML-encoded.
Description	string	''	Supporting copy under the title, HTML-encoded.

Visual

What it does: *Icon* and *ImageURL* supply the panel's visual. Set one or the other, not both: when **ImageURL** is non-empty it wins over **Icon** outright, because a picture placeholder and an icon placeholder on the same panel is never what the caller meant.

Icon accepts either of two shapes, and the component tells them apart by looking only at the first non-blank character of the string. If that character is not '<', the whole string is treated as a CSS icon-font class, for example `bi bi-inbox`, and rendered inside a wrapper: `<div class="sgc-empty-visual"><i class="bi bi-inbox"></i></div>`. If the first non-blank character is '<', the whole string is treated as raw inline `<svg>...</svg>` markup and, once it passes the guard described below, is emitted byte for byte inside the same wrapper div.

Security warning: never assign end-user input to Icon. Assign it to **Title** / **Description** instead, which are HTML-encoded. Icon markup is emitted verbatim, not encoded.

Because inline markup is emitted verbatim, it first passes through a narrow guard, the internal function `sgcEmptyStateSafeMarkup`. The guard rejects the icon outright, dropping it completely rather than trying to clean it up, when the markup contains a `<script`, `<iframe`, `<foreignObject`, `<object` or `<embed` tag, a `javascript:` URL, or any attribute whose name starts with "on" immediately followed by "=" (a generic event-handler detector, not a fixed list of onclick/onerror/etc; it walks the string character by character looking for that `on<name>=` shape after a space, tab, newline, carriage return, '/', "" or """). When the guard rejects the markup, **Icon** contributes nothing to the output at all, no wrapper div, rather than a stripped or half-cleaned fragment.

That full-rejection choice is deliberate: rejection drops the whole icon rather than trying to clean it up, because a half-sanitized fragment is the shape that keeps coming back as the next bypass. Sanitizing markup down to something that looks safe invites the next attacker to find the one tag or attribute the sanitizer missed; refusing to render anything the guard cannot fully vouch for closes that door instead of leaving it ajar.

ImageURL goes through the same URL sanitizer used across sgcHTML, with `data:` image URLs additionally allowed alongside http(s)/mailto/tel, since an inline <img> is a common way to hand the component a small embedded picture; any other scheme, or a data: URL whose media type is not image/*, falls back to a blank source. It renders as `<img src="..." alt="" class="sgc-empty-image">`.

Property	Type	Default	Description
Icon	string	''	CSS icon-font class or inline SVG markup for the panel's visual.
ImageURL	string	''	Image shown instead of Icon; wins over Icon when both are set.

Actions

What it does: *ActionCaption* / *ActionHref* / *ActionStyle* render the panel's primary call-to-action link, and only when *ActionCaption* is non-empty. *SecondaryActionCaption* / *SecondaryActionHref* render a second link alongside it, again only when its caption is non-empty.

The primary action is styled through the shared **TsgcHTMLButtonStyle** enumeration (default `bsPrimary`), the same button-style type used across the sgcHTML components. The secondary action has no style property of its own: it is always rendered with the `btn btn-outline-secondary` class, on the reasoning that a second, less prominent action next to a primary one reads correctly as an outline button in every Bootstrap theme without asking the caller to pick a style for it.

Both hrefs go through the same URL sanitizer as **ImageURL** (without the data: image allowance), and both fall back to `href="#"` when the value is rejected, rather than omitting the link the way **Icon** is dropped outright when it fails its guard. The two properties can afford different failure behavior because they fail differently: a href is always attribute-encoded before being written into the page, so even a rejected value can only ever become a harmless "#" link, never executable markup, whereas Icon markup is emitted completely unencoded and a bad fragment that slipped through would be executable. Since **ActionCaption** was explicitly set, silently dropping the link would be a worse surprise than leaving a visible button that points nowhere, so the fallback keeps the button on the page with a safe target instead.

Property	Type	Default	Description
ActionCaption	string	″	Caption of the primary call-to-action link; empty renders no action.
ActionHref	string	″	Target of the primary call-to-action link, sanitized on output.
ActionStyle	TsgcHTML-ButtonStyle	bsPri-mary	Bootstrap button style of the primary action.
SecondaryAc-tionCaption	string	″	Caption of the secondary call-to-action link; empty renders no secondary action.
SecondaryAc-tionHref	string	″	Target of the secondary call-to-action link, sanitized on output.

Layout

What it does: *Compact* switches the panel to tighter padding, meant for a small embedded area such as a widget or a table body, as opposed to the default full-size padding meant for a whole empty page or section. *Bordered* adds a `border rounded` class around the panel. *TextAlign* controls how the title, description and actions are aligned within it.

TextAlign is a **TsgcHTMLTextAlign** value shared across sgcHTML: `taLeft` aligns to the start of the panel, `taCenter` (the default) centers everything, and `taEnd` aligns to the end. Center is the default because most empty states are the sole content of their container and read best centered rather than pinned to one side.

Property	Type	Default	Description
Compact	Boolean	False	Switches to tighter padding for a small embedded area.
Bordered	Boolean	False	Adds a bordered, rounded frame around the panel.
TextAlign	TsgcHTMLTextAlign	taCenter	Alignment of the title, description and actions.

Output

The inherited **HTML** property returns a single `<div class="sgc-empty-state">` (with `sgc-empty-compact`, `border rounded` and a text-alignment class such as `text-center` appended as the corresponding properties are set), containing, in order, the visual, the title, the description and the actions wrapper, whichever of those are non-empty. No `<script>` is ever part of the output.

Reference

- Properties — 12 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_ProgressBar

TsgcHTMLComponent_ProgressBar renders a Bootstrap progress bar, either a single filled bar or a stack of colored segments built from the Bars collection.

Introduction

TsgcHTMLComponent_ProgressBar emits a Bootstrap `<div class="progress">` containing one or more `<div class="progress-bar">` segments. In single-value mode it draws one segment sized by **Value** and **Max**, with the color, stripes, animation and label text you choose. Read the finished markup through the inherited **HTML** property.

For a multi-segment bar, for example disk usage broken down by category, add items to the **Bars** collection. As soon as Bars has one or more items the component switches to stacked mode: the component-level *Value*, *Color*, *ColorStyle* and *ShowLabel* are ignored, and each **TsgcHTMLProgressBarItem** supplies its own *Value*, *Color*, *ColorStyle* and *Label_* instead.

Quick Start

1. Drop a **TsgcHTMLComponent_ProgressBar** component on the form (or create it at runtime).

2. Set **Value** and **Max** for a single bar, or fill the **Bars** collection for a stacked bar.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_ProgressBar *oProgressBar = new TsgcHTMLComponent_ProgressBar(NULL);
try
{
  oProgressBar->Value = 65;
  oProgressBar->Max = 100;
  oProgressBar->ColorStyle = hcSuccess;
  oProgressBar->ShowLabel = true;
  html = oProgressBar->HTML;
}
__finally
{
  delete oProgressBar;
}
```

Value and Range

What it does: *Value* and *Max* control the fill percentage in single-value mode. The percentage is `Round(Value * 100 / Max)`, clamped to the 0..100 range. A *Max* of zero or less is treated as 100.

Property	Type	Default	Description
Value	Integer	0	Current amount in single-value mode; ignored once Bars has items.

Max	Integer	100	Amount that corresponds to a full (100%) bar; values of 0 or less are treated as 100.

Appearance

What it does: *Color*, *ColorStyle*, *Striped* and *Animated* style the single bar; *Height* sets the outer bar's thickness and *ProgressBarID* assigns it an id. The coloring and texture properties are ignored once **Bars** has items, because each stacked segment supplies its own.

Property	Type	Default	Description
Color	string	''	Raw CSS color (for example #7C3AED) applied as an inline background-color; overrides ColorStyle when set.
ColorStyle	TsgcHTMLColor	hcPrimary	Bootstrap contextual color (bg-primary, bg-success, ...) used when Color is empty.
Striped	Boolean	False	Adds diagonal stripes to the bar (progress-bar-striped).
Animated	Boolean	False	Scrolls the stripes continuously; implies Striped even when Striped is False.
Height	string	''	CSS height (for example 24px) of the outer .progress element; empty keeps Bootstrap's default height.
ProgressBarID	string	''	id of the outer div, used only when the inherited ElementID is empty.

Label

What it does: *ShowLabel* and *LabelFormat* draw the percentage as text inside the bar in single-value mode. In stacked mode, use each segment's own *Label_* instead.

Property	Type	Default	Description
ShowLabel	Boolean	False	Draws the formatted percentage text inside the bar.
LabelFormat	string	'%d%%'	Format() string applied to the integer percent; falls back to plain "n%" text if invalid.

Stacked Bars

What it does: The **Bars** collection holds the segments of a stacked bar. Call **Bars.Add** once per segment and set its sub-properties. As soon as Bars has one or more items, the component renders one `.progress-bar` per segment instead of the single component-level bar. Each entry is a **TsgcHTMLProgressBarItem** exposing the following published properties.

Bar Property	Type	Default	Description
Value	Integer	0	Amount for this segment, measured against the component's Max.
Color	string	''	Raw CSS color for this segment; overrides ColorStyle when set.
ColorStyle	TsgcHTMLColor	hcPrimary	Bootstrap contextual color for this segment, used when Color is empty.
Label_	string	''	Text rendered inside this segment.

```
TsgcHTMLProgressBarItem *bar;
bar = oProgressBar->Bars->Add(); bar->Value = 40; bar->ColorStyle = hcSuccess; bar->Label_ = "Used";
bar = oProgressBar->Bars->Add(); bar->Value = 20; bar->ColorStyle = hcWarning; bar->Label_ = "Pending";
bar = oProgressBar->Bars->Add(); bar->Value = 10; bar->ColorStyle = hcDanger; bar->Label_ = "Failed";
```

Output

The inherited **HTML** property returns the outer `<div class="progress">` (carrying the id and, when set, the Height style) wrapping one or more `<div class="progress-bar">` elements, each with `role="progressbar"` and `aria-valuenow` / `aria-valuemin` / `aria-valuemax` attributes. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 11 properties available on the component.
- Methods — this component has no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_Badge

TsgcHTMLComponent_Badge renders a small Bootstrap badge, an inline label used to show a count, a status word or a short tag next to other content.

Introduction

TsgcHTMLComponent_Badge emits a single `<span>` styled with Bootstrap's `badge` classes. You set the **Text** to display, pick a theme **Color**, and optionally round it into a pill or add a leading **Icon**. The complete markup is returned through the inherited **HTML** property.

TsgcHTMLComponent_Badge shares its unit with TsgcHTMLComponent_Chip, a related component that renders a dismissible tag with an avatar or icon. Use a badge for a short static label or counter; use a chip when the label needs an avatar image or a close button.

Quick Start

1. Drop a **TsgcHTMLComponent_Badge** component on the form (or create it at runtime).

2. Set the **Text** and a **Color**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Badge *oBadge = new TsgcHTMLComponent_Badge(NULL);
try
{
  oBadge->Text = "New";
  oBadge->Color = bgSuccess;
  oBadge->Pill = true;
  html = oBadge->HTML;
}
__finally
{
  delete oBadge;
}
```

Text and Icon

What it does: *Text* is the label shown inside the badge; it is HTML-encoded automatically, so it is safe to pass user-supplied strings. *Icon* optionally draws a CSS icon (for example a Bootstrap Icons class) before the text.

Property	Type	Default	Description
Text	string	''	The label drawn inside the badge. HTML-encoded when rendered.
Icon	string	''	CSS icon class drawn before the text, for example 'bi bi-check'. Leave empty for no icon.

Color and Shape

What it does: *Color* selects the Bootstrap theme variant used for the badge background. *Pill* rounds the badge into a fully rounded pill shape instead of the default slightly rounded rectangle.

Property	Type	Default	Description
Color	TsgcHTMLBadgeStyle	bgPrimary	Theme variant: bgPrimary, bgSecondary, bgSuccess, bgDanger, bgWarning, bgInfo, bgLight or bgDark.
Pill	Boolean	False	When True the badge is rounded into a pill shape (adds the rounded-pill class).

Positioned Badges (counters)

What it does: *Positioned* turns the badge into a notification counter that overlays the top-right corner of its parent element (for example a button or an icon), the way an unread-message count sits on top of a bell icon. The parent element needs `position: relative` so the absolute positioning is relative to it, not to the page.

Property	Type	Default	Description
Positioned	Boolean	False	When True the badge is absolutely positioned over the top-right corner of its parent, for notification-counter badges. The parent needs position: relative.

```
// Inline badge inside another component's markup, with no component kept around
row = "<li>Inbox " + TsgcHTMLComponent_Badge::Build("4", bgDanger, true) + "</li>";
```

Output

The inherited **HTML** property returns a single `<span class="badge bg-<variant>">` element, with `rounded-pill` and the position-absolute classes added when **Pill** or **Positioned** are set. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 5 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_Chip

TsgcHTMLComponent_Chip renders a chip, a pill-shaped tag that can carry an avatar image, an icon and an optional close button, used for filters, selections and removable tags.

Introduction

TsgcHTMLComponent_Chip emits a single `<span>` styled as a rounded Bootstrap badge that lays out an optional avatar picture, an optional icon, the chip's **Text** and, when **Dismissible** is set, a small close button. The complete markup is returned through the inherited **HTML** property.

TsgcHTMLComponent_Chip shares its unit with TsgcHTMLComponent_Badge, a related, simpler component for static labels and counters. Use a badge for a short static label or counter; use a chip when the label needs an avatar, an icon, or a way for the user to remove it.

Quick Start

1. Drop a **TsgcHTMLComponent_Chip** component on the form (or create it at runtime).

2. Set the **Text**, a **Color**, and **Dismissible** if the user should be able to remove it.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Chip *oChip = new TsgcHTMLComponent_Chip(NULL);
try
{
  oChip->Text = "JavaScript";
  oChip->Icon = "bi bi-tag";
  oChip->Color = bgInfo;
  oChip->Dismissible = true;
  html = oChip->HTML;
}
__finally
{
  delete oChip;
}
```

Content (text, icon, avatar)

What it does: *Text* is the chip's label, HTML-encoded automatically. *Icon* optionally draws a CSS icon before the text. *AvatarURL* optionally draws a small round picture before the icon and text, useful for a chip that represents a person. Content is laid out in the order avatar, icon, text.

Property	Type	Default	Description
Text	string	''	The label drawn on the chip. HTML-encoded when rendered.

Icon	string	''	CSS icon class drawn before the text, for example 'bi bi-tag'. Leave empty for no icon.
AvatarURL	string	''	URL of a small round avatar picture (1.25em, object-fit: cover) drawn before the icon and text. The URL is sanitized. Leave empty for no avatar.

Color and Style

What it does: *Color* selects the Bootstrap theme variant. *Outline* switches the chip from a solid filled background to a transparent background with a colored border and text, a lighter look often used for filter chips.

Property	Type	Default	Description
Color	TsgcHTMLBadgeStyle	bgSecondary	Theme variant: bgPrimary, bgSecondary, bgSuccess, bgDanger, bgWarning, bgInfo, bgLight or bgDark.
Outline	Boolean	False	When True the chip is drawn with a transparent background and a colored border and text instead of a solid fill.

Dismissible Chips

What it does: *Dismissible* adds a small close button at the end of the chip. Clicking it removes the chip element from the page immediately, entirely in the browser; nothing is sent back to the server. On the solid, dark variants the close button is also drawn in white so it stays visible against the background.

Property	Type	Default	Description
Dismissible	Boolean	False	When True a close button is added; clicking it removes the chip from the DOM on the client. Note the Build helper's aDismissible parameter defaults to True, unlike this property.

Because removal happens purely on the client, the server is never told that a chip was dismissed. If your application needs to know, wrap the chips in a form and submit the remaining selection, or attach your own script to the close button before it removes the element.

```
// One-line render with the static helper; note aDismissible defaults to true here
html = TsgcHTMLComponent_Chip::Build("React", bgInfo);
```

Output

The inherited **HTML** property returns a single `<span class="badge rounded-pill d-inline-flex align-items-center sgc-chip bg-<variant>">` element (or the `bg-transparent border ... text-<variant>` variant when **Outline** is True), containing the avatar image, icon, text and close button that apply. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 6 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_ContextMenu

TsgcHTMLComponent_ContextMenu renders a hidden Bootstrap dropdown menu that opens at the mouse pointer on a right-click, wired entirely by an inline script.

Introduction

TsgcHTMLComponent_ContextMenu emits a hidden Bootstrap `<ul class="dropdown-menu shadow">` built from the **Items** collection, plus a small inline script. The script listens for the browser's `contextmenu` event; when **TargetSelector** is set the menu only opens if the click target matches that CSS selector (using `closest()`), otherwise it opens anywhere on the page. It positions the menu at the pointer, clamps it to the viewport, and hides it again on an outside click, a click on an item, or the Escape key.

Each entry is a **TsgcHTMLContextMenuItem** exposing *Caption*, *Icon*, *Href*, *DataAction*, *Divider*, *Header* and *Disabled*. Give an item a non-empty *DataAction* to route its click to the server over the WebSocket / HTMX transport instead of a plain link.

Quick Start

1. Drop a **TsgcHTMLComponent_ContextMenu** component on the form (or create it at runtime).

2. Add one item per command to the **Items** collection; optionally set **TargetSelector** to scope the menu to matching elements.

3. Read the **HTML** property and add it to the page once; the script handles opening, positioning and hiding.

```
TsgcHTMLComponent_ContextMenu *oContextMenu = new TsgcHTMLComponent_ContextMenu(NULL);
try
{
  oContextMenu->TargetSelector = ".grid-row";
  TsgcHTMLContextMenuItem *item;
  item = oContextMenu->Items->Add(); item->Caption = "Edit"; item->Icon = "bi bi-pencil"; item->DataAction =
  item = oContextMenu->Items->Add(); item->Caption = "Delete"; item->Icon = "bi bi-trash"; item->DataAction =
  item = oContextMenu->Items->Add(); item->Divider = true;
  item = oContextMenu->Items->Add(); item->Caption = "Properties"; item->DataAction = "props";
  html = oContextMenu->HTML;
}
__finally
{
  delete oContextMenu;
}
```

Menu Items

What it does: The **Items** collection holds one entry per command, in the order they should appear. Call **Items.Add** to append an entry and set its sub-properties. Set *Divider* to draw a separator instead of a command, or *Header* to draw a non-clickable section heading; *Disabled* greys out a command and blocks clicks on it. Each entry is a **TsgcHTMLContextMenuItem** exposing the following published properties.

Item Property	Type	Default	Description
Caption	string	''	Visible text of the entry (or of a Header).
Icon	string	''	CSS icon class, for example bi bi-pencil, rendered as a leading <i> element.
Href	string	''	Link target, sanitized on output; empty renders href="#".
DataAction	string	''	Action id sent to the server when the entry is clicked; see Actions over WebSocket below.
Divider	Boolean	False	Renders a <hr class="dropdown-divider"> instead of a command.
Header	Boolean	False	Renders a non-clickable <h6 class="dropdown-header"> instead of a command.
Disabled	Boolean	False	Greys out the entry, adds tabindex="-1" and blocks clicks on it.

```
TsgcHTMLContextMenuItem *item;
item = oContextMenu->Items->Add(); item->Header = true; item->Caption = "Row Actions";
item = oContextMenu->Items->Add(); item->Caption = "Duplicate"; item->DataAction = "duplicate";
item = oContextMenu->Items->Add(); item->Caption = "Archive"; item->DataAction = "archive"; item->Disabled =
```

Where the Menu Opens

What it does: *TargetSelector* restricts the right-click zone to elements matching a CSS selector; leave it empty to let the menu open anywhere on the page. *MenuID* sets the id of the generated <ul>, which the inline script also uses (through `document.getElementById`) to find its own menu element.

Leave *MenuID* empty (the default) and a unique element id is generated automatically the first time the component renders, and reused for the life of the instance, so two context menus on the same page never collide and the id stays stable across re-renders and HTMX out-of-band swaps. Set it explicitly if you need a stable, known id, for example to reach the menu from your own JavaScript or to recognize on the server which menu raised a *DataAction*.

Property	Type	Default	Description
TargetSelector	string	''	CSS selector (matched with closest()) that the click target must be inside; empty opens the menu anywhere on the page.
MenuID	string	(empty)	id of the generated element. Empty means a unique auto id, generated once per instance.

Actions over WebSocket

What it does: Any item with a non-empty *DataAction* becomes a server action instead of a plain link. Clicking it does not navigate; the emitted script builds the flat JSON payload `{action:"...", menu:"..."}` and sends it with `window.sgcHTMX.send(...)`, the same transport used by `data-sgc-ws-send` forms, so the server's WebSocket / HTMX engine receives it exactly like a form submission. The script also dispatches a DOM `CustomEvent` named `sgcContextMenu:action` (its `detail` is the same payload), so a page can react purely client-side without a round trip. Disabled items ignore clicks, and the menu hides itself after any item is clicked.

Output

The inherited **HTML** property returns the hidden `<ul class="dropdown-menu shadow">` together with the inline script that opens, positions and hides it. Add the returned markup to the page once, for example near the end of a layout template; no further calls are required afterward. Serve it directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap**.

Reference

- Properties — 3 properties available on the component.
- Methods — this component has no public methods of its own.
- Events — this component has no published events.

TsgcHTMLComponent_CommandPalette

TsgcHTMLComponent_CommandPalette renders a hidden Ctrl+K / Cmd+K overlay command launcher, the quick action search popularized by VS Code, Slack, Linear and Spotlight, wired entirely by an inline script.

Introduction

TsgcHTMLComponent_CommandPalette emits a hidden overlay dialog built from the **Items** collection, plus a small inline script. The overlay starts hidden and is opened by the configured hotkey, and closed again by Esc, a click on the backdrop, or by activating an entry. Typing in the search input filters the entries in two passes: a substring match first, then a subsequence ("fuzzy") pass, so typing `nu` still finds "New User" even though `nu` is not a contiguous substring of it. The arrow keys (and Home / End) move the selection, and Enter activates the highlighted entry.

The entries are rendered as markup on the server, not shipped to the browser as a JSON array to filter client-side: every *Caption*, *Description*, *Category* and *Shortcut* is HTML-encoded exactly once, server-side, and the client-side filter only ever reads back the `data-sgc-text` attribute the server already wrote, it never re-renders anything. *Href* goes through `sgcHTMLSanitizeURL`: a `javascript:` URL is rejected and the entry silently degrades to a `<button>` instead of a dead or dangerous link, which is the fix the Notification component needed after the same bug shipped there.

Two palettes on the same page do not fight over the hotkey: the global keydown handler refuses to open a second palette while another one is already open (checked with `document.querySelector('.sgc-cmdp-open')`), so the second hotkey press is simply ignored until the first palette is closed.

Quick Start

1. Drop a **TsgcHTMLComponent_CommandPalette** component on the form (or create it at runtime).

2. Add one item per command to the **Items** collection with **AddItem**, or fill it from a dataset with **LoadFromDataSet**.

3. Read the **HTML** property and add it to the page once; the script handles opening the overlay on Ctrl+K / Cmd+K, filtering, keyboard navigation and closing.

```
TsgcHTMLComponent_CommandPalette *oPalette = new TsgcHTMLComponent_CommandPalette(NULL);
try
{
  oPalette->AddItem("New User", "", "new_user", "Create", "Ctrl+N");
  oPalette->AddItem("New Invoice", "", "new_invoice", "Create");
  TsgcHTMLCommandItem *item = oPalette->AddItem("Settings", "/settings", "", "Navigate");
  item->Icon = "bi bi-gear";
  html = oPalette->HTML;
}
__finally
{
  delete oPalette;
}
```

Palette Entries

What it does: The **Items** collection holds one entry per command, in the order they should appear. Call **AddItem** to append an entry and get back the created **TsgcHTMLCommandItem** so its remaining sub-properties can still be set, or call **LoadFromDataSet** to fill the whole collection from a dataset in one call.

Property	Type	Default	Description
Items	TsgcHTMLCommandItems	(empty collection)	Collection of TsgcHTMLCommandItem entries shown in the palette.

Each entry is a **TsgcHTMLCommandItem** exposing the following published properties.

Item Property	Type	Default	Description
Caption	string	''	Visible text of the entry.
Description	string	''	Secondary line shown smaller and muted, under the caption.
Icon	string	''	CSS icon-font class, for example bi bi-gear, rendered as a leading <i> element.
Href	string	''	When set the entry is a link, subject to sgcHTMLSanitizeURL; a rejected scheme degrades the entry to a button.
DataAction	string	''	Opaque command name delivered to the page in the sgc:command event, and posted as the "command" field when Action / WSSend is used.
Shortcut	string	''	Keyboard-shortcut hint shown right-aligned in a <kbd>, for example Ctrl+S; display only, it does not bind an actual key.
Category	string	''	Groups entries under a heading when ShowCategories is True.
Enabled	Boolean	True	False greys the row out, adds aria-disabled="true" and tabindex="-1", and blocks the click.

Shortcut is decoration only. It is rendered in a `<kbd>` element on the right of the row so the user can see the keyboard shortcut a command is bound to elsewhere in the page, but the palette itself never listens for that key combination, so nothing breaks if the page has not actually wired that shortcut up.

Category grouping preserves first-appearance order, not alphabetical order. When *ShowCategories* is True, the groups are emitted in the order their category first appears among the items, not sorted alphabetically, so the caller controls the layout entirely by the order items are added to the collection.

```
oPalette->AddItem("New User", "", "new_user", "Create");
TsgcHTMLCommandItem *item = oPalette->Items->Add();
item->Caption = "Invite Teammate";
item->Description = "Send an email invitation";
item->Icon = "bi bi-person-plus";
item->DataAction = "invite";
item->Category = "Create";
item->Enabled = UserIsAdmin;
```

LoadFromDataSet clears *Items* first, then walks the dataset from its first record to end of file, filling only the fields that were actually named.

```
oPalette->LoadFromDataSet(qryCommands, "Caption", "Description", "Href",
  "ActionName", "Category", "Shortcut", "Icon");
```

Opening the Palette (Hotkey)

What it does: *PaletteID* sets the id of the generated overlay element, and *HotKey* / *HotKeyCtrl* / *HotKeyMeta* control the keyboard shortcut that opens it.

Property	Type	Default	Description
PaletteID	string	(empty)	id of the generated overlay element. Empty means a unique auto id, generated once per instance.
HotKey	string	'k'	Single alphanumeric character that opens the palette, combined with HotKeyCtrl / HotKeyMeta.
HotKeyCtrl	Boolean	True	The Ctrl modifier opens the palette when True.
HotKeyMeta	Boolean	True	The Cmd / Meta modifier (macOS) opens the palette when True.

Leave *PaletteID* empty (the default) and a unique element id is generated automatically the first time the component renders, and reused for the life of the instance, the same auto-id pattern used by every other sgcHTML component. Set it explicitly if you need a stable, known id.

HotKey validation. Only a single alphanumeric character is accepted: a value that is empty, longer than one character, or not a letter or digit is silently ignored and the palette falls back to its default hotkey, `k`. There is no design-time error for an invalid value, so a typo in *HotKey* quietly keeps the default rather than breaking the component.

Ctrl+K and Cmd+K both work out of the box. *HotKeyCtrl* and *HotKeyMeta* both default to True, and the opener script checks `(HotKeyCtrl && ctrlKey) || (HotKeyMeta && metaKey)`, so either Ctrl+K on Windows/Linux or Cmd+K on macOS opens the palette without any extra configuration, which is exactly the cross-platform convenience the quick action search pattern is known for. Set one of the two to False if you need the palette to respond to only that specific modifier.

```
oPalette->HotKey = "p";
oPalette->HotKeyCtrl = true;
oPalette->HotKeyMeta = false; // Windows/Linux only: Ctrl+P
```

Search & Display

What it does: *Placeholder* and *EmptyText* control the search input's hint text and the "no results" message, *MaxResults* caps how many matching entries are shown at once, and *ShowCategories* / *ShowShortcuts* control whether the entries are grouped and whether the shortcut hints are drawn.

Property	Type	Default	Description
Placeholder	string	(locale)	Search input placeholder; empty resolves to the active locale's search text at render time.
EmptyText	string	(locale)	Shown when the filter matches nothing; empty resolves to the active locale's "no results" text.
MaxResults	Integer	10	Maximum number of entries shown at once; 0 means show all matching entries.
ShowCategories	Boolean	True	Groups the list under headings by Category; False renders one flat ungrouped list.
ShowShortcuts	Boolean	True	Shows the <kbd> shortcut hint column when an item has a Shortcut.

When left empty, *Placeholder* comes from `sgcHTMLDefaultLocale.SearchText` (English: "Search") and *EmptyText* comes from `sgcHTMLDefaultLocale.MultiSelectNoResultsText` (English: "No results"), both from unit `sgcHTML_Locale`. The fallback is resolved on every render, not in the constructor, so a per-request locale override is honoured even by a single long-lived component instance.

```
oPalette->Placeholder = "Type a command or search...";
oPalette->EmptyText = "Nothing matched your search";
oPalette->MaxResults = 8;
oPalette->ShowCategories = true;
```

Actions on Activation

What it does: Activating an entry (a click, or Enter on the highlighted row) always dispatches a DOM `CustomEvent` named `sgc:command` (its `detail` is `{id, action, caption}`), the same client-only reaction pattern used by the ContextMenu component's `sgcContextMenu:action` event. When *Action* or *WSSend* is also set, and the activated entry is not a plain link (it has no *Href*), the activated command is additionally sent to the server: as a WebSocket push when *WSSend* is True, or as a POST to *Action* otherwise.

Property	Type	Default	Description
Action	string	''	When set, and WSSend is False, the activated command is POSTed here as the "command" field.
WSSend	Boolean	False	When True the activated command is pushed over the sgcHTMX WebSocket bridge instead of being POSTed, the same transport used by a data-sgc-ws-send form.

A hidden form is only emitted when *WSSend* is True or *Action* is non-empty. An entry that has its own *Href* is a real link: activating it keeps its own default browser navigation, and it is never submitted as a command, since a link that also fired a server command would leave the page mid-navigation. An entry with no *Href* fills the form's hidden

"command" field with its *DataAction* and submits the form, either through `window.sgcHTMX` (WSSend True) or as a normal POST to *Action*.

```
oPalette->WSSend = true; // push the activated command over the WebSocket
// or, to POST it instead:
// oPalette->Action = "/commands/run";
```

Output

The inherited **HTML** property returns the hidden overlay markup together with the inline script that opens it on the hotkey, filters and navigates the list, and closes it again. Add the returned markup to the page once, for example near the end of a layout template; no further calls are required afterward. Serve it directly, or assign it to the Body-Content of a **TsgcHTMLTemplate_Bootstrap**.

Reference

- Properties — 12 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Presence

TsgcHTMLComponent_Presence renders a "who is online" roster, either as a stack of avatars or as a list, and updates a single user in the browser by pushing one small fragment over the WebSocket connection.

Introduction

TsgcHTMLComponent_Presence keeps a collection of users, each with an id, a display name, an optional avatar and a status (online, away, busy or offline). It renders them either as a stack of overlapping avatar circles (`plAvatars`) or as a vertical list with names and status text (`plList`), with an optional title and an online-count badge. The complete markup is returned through the inherited **HTML** property.

The roster is rendered on the server, in Delphi. Nothing about it is computed in the browser: there is no client-side data binding, no polling timer and no JSON payload to parse.

That matters because a presence roster changes constantly. When one user goes away or signs out, you do not want to re-render the page for every connected browser. The component solves this by rendering just that one user and letting you push the result over the WebSocket connection the page already has open. See **Live Updates over WebSocket** below.

Quick Start

1. Drop a **TsgcHTMLComponent_Presence** component on the form (or create it at runtime).

2. Add the users that are currently connected with **AddUser**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Presence *oPresence = new TsgcHTMLComponent_Presence(NULL);
try
{
  oPresence->PresenceID = "roster";
  oPresence->Title = "Online now";
  oPresence->AddUser("u1", "Ana Ruiz", psOnline, "/img/ana.png");
  oPresence->AddUser("u2", "Mark Fisher", psBusy, "", "In a meeting");
  html = oPresence->HTML;
}
__finally
{
  delete oPresence;
}
```

Later, when one of those users changes status, do not rebuild the page. Change the user and push that user only.

```
oPresence->SetUserStatus("u2", psAway, "Back in 10");
oEngine->BroadcastFragment(oPresence->GetUserFragmentHTML("u2"));
```

Users and Status

What it does: *Users* is a collection of **TsgcHTMLPresenceUser** items. You normally fill it at runtime with **AddUser**, **SetUserStatus** and **RemoveUser** rather than editing the collection directly, but the items can also be created at design time in the Object Inspector.

Item Property	Type	Default	Description
Id	string	''	Your key for the user (a session id, a login, a database key). It is what every method takes, and it is what the element id of the user's row is derived from.
DisplayName	string	''	The name shown to other users. When AvatarURL is empty, the component draws the initials taken from this name instead of a picture.
AvatarURL	string	''	Picture for the avatar circle. Leave it empty to fall back to initials.
Status	TsgcHTMLPresenceStatus	psOnline	psOnline, psAway, psBusy or psOffline. The status drives the color of the small dot drawn on the avatar: green, amber, red and grey respectively.
StatusText	string	''	Free text shown instead of the default status word ("Online", "Away", "Busy", "Offline"), for example "In a meeting".

The status word (or StatusText) is used in two places: in `plList` layout it is printed under the display name when **ShowStatusText** is True, and in `plAvatars` layout it is always written into the avatar's `title` tooltip.

GetOnlineCount counts every user whose status is not `psOffline`, so away and busy users still count as present. That count is what the badge shows.

Live Updates over WebSocket

What it does: this is the point of the component. When one user changes, you do not re-render the page and you do not poll. You render that one user and push the result to the browsers that are already connected.

Every user row is written with a stable element id, built from **PresenceID** plus the user's *Id*, and **every** user in the collection is rendered (the *MaxVisible* cap only hides the overflow with CSS, it does not leave them out of the markup). So every user has an element in the page, and a fragment pushed for any of them, even one currently beyond the cap, always lands. The "+N" overflow chip, the online-count badge and the empty-state placeholder each have their own stable id too. **GetUserFragmentHTML** returns the markup of that one user marked as an htmx out-of-band swap (`hx-swap-oob`), which means: when this markup reaches the browser, replace the element with that id, wherever it is on the page, and leave everything else alone.

The fragment is sent to the browser through the WebSocket connection the page already holds, using the sgcHTML engine:

Engine Method	Description

`procedure PushFragment(const aGuid, aHTML: string);`	Sends the fragment to a single client, addressed by its connection guid.
`procedure BroadcastFragment(const aHTML: string; const aChannel: string = ''; const aExclude: string = '');`	Sends the fragment to every connected client, optionally only those on a channel, optionally excluding one guid.

The browser applies the swap in place. There is no HTTP round-trip, no polling and no client-side data binding: the server owns the roster and pushes only the difference.

The flow is always the same, whatever changed: **mutate the roster, then push**. Removal works with the very same call, because when the user no longer exists **GetUserFragmentHTML** returns a delete fragment for that element id instead of markup, and the browsers drop the row.

The counters and the empty state ride along. A single call to **GetUserFragmentHTML** returns the user row (or delete fragment) *and* three further out-of-band fragments: the "+N" overflow chip, the online-count badge and the empty-state placeholder. Any change to a user can move all three, a status turning offline changes the count, a removal changes the overflow, and removing the last user has to bring the empty text back, so they all travel in the one push. You do not need to push **GetCountFragmentHTML** or **GetMoreFragmentHTML** separately; those methods are there for the odd case where only a counter changed. When the host empties the roster in one go, push **GetEmptyFragmentHTML** to re-render the placeholder on its own.

```
// a user signed in: one push carries the row, the chip, the count and the empty state
void __fastcall TServerForm::UserConnected(const String aId, const String aName)
{
  oPresence->AddUser(aId, aName, psOnline);
  oEngine->BroadcastFragment(oPresence->GetUserFragmentHTML(aId));
}

// a user signed out: RemoveUser first, then the SAME push removes the row
void __fastcall TServerForm::UserDisconnected(const String aId)
{
  oPresence->RemoveUser(aId);
  oEngine->BroadcastFragment(oPresence->GetUserFragmentHTML(aId));
}
```

Use **PushFragment** instead of **BroadcastFragment** when only one browser should see the change, and pass a channel to **BroadcastFragment** when the roster is per room rather than global.

Layout and Appearance

What it does: *Layout* chooses between the avatar stack and the list. *MaxVisible* caps how many users are *shown* before the rest collapse into a "+N" overflow badge (every user is still rendered, the cap is applied with CSS). *AvatarSize*, *Title*, *ShowCount*, *ShowStatusText* and *EmptyText* tune the rest.

Property	Type	Default	Description
Users	TsgcHTMLPresenceUsers	Empty collection	The roster: a collection of TsgcHTMLPresenceUser items.
PresenceID	string	''	The id of the generated root element, and the prefix of every user element id. Leave it empty and a unique id is generated on the first render, so two rosters on one page never collide.

Layout	TsgcHTMLP-resenceLayout	plAvatars	plAvatars draws a stack of overlapping avatar circles; plList draws one row per user with name and status.
MaxVisi-ble	Integer	5	Number of users shown before the roster collapses the rest into a "+N" overflow badge. Every user is still ren-dered (the cap is CSS-only), so a push for any user always lands. Set 0 to show all of them.
ShowSta-tusText	Boolean	True	In plList layout, prints the status line under the display name.
Show-Count	Boolean	True	Draws the online-count badge next to the title.
Title	string	'Online'	Heading drawn above the roster. Empty hides it.
Avatar-Size	Integer	40	Diameter in pixels of the avatar circle. Values below 16 are clamped to 16.
Empty-Text	string	'Nobody is online'	Text drawn when the roster is empty.

Output

The inherited **HTML** property returns a `<div>` carrying the **PresenceID**, an optional header (title plus count badge) and an inner `<div>` holding the users, one element per user, each with its own stable id, followed by the "+N" over-flow chip and the empty-state placeholder (each hidden when it does not apply). When **MaxVisible** is set, an inline `<style>` scoped to this roster hides the users past the cap by position. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document. The component emits no script of its own: the roster is plain markup, and it only changes when you push a fragment to it.

Display names, status text and avatar URLs are encoded and sanitized before they are written, so user-supplied values cannot inject markup.

Reference

- Properties — 9 properties available on the component.
- Methods — 8 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_ActivityFeed

TsgcHTMLComponent_ActivityFeed renders a live, append-only "who did what, when" feed, kept in a ring buffer and grown in the browser by pushing one small fragment per new entry over the WebSocket connection.

Introduction

TsgcHTMLComponent_ActivityFeed keeps a collection of activity items, each one an actor, an action, a target and a timestamp ("Ana updated invoice #1042, 3 minutes ago"), plus an optional icon, colour and avatar. It renders them newest first, with a title and an empty-state message, and returns the complete markup through the inherited **HTML** property.

The feed is rendered on the server, in Delphi. When something happens you add one item and push one fragment: you do not re-render the page, and the browser does not poll.

Items are held in a **ring buffer** capped by **MaxItems**. Adding an item beyond the cap drops the oldest one, so a long-running server does not accumulate history it will never show. See **Activity Items** and **MaxItems** for what exactly that means for the browser.

Quick Start

1. Drop a **TsgcHTMLComponent_ActivityFeed** component on the form (or create it at runtime).

2. Record something that happened with **AddActivity**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_ActivityFeed *oFeed = new TsgcHTMLComponent_ActivityFeed(NULL);
try
{
  oFeed->FeedID = "feed";
  oFeed->Title = "Recent activity";
  oFeed->AddActivity("Ana", "updated", "invoice #1042", hcSuccess, "bi bi-pencil");
  html = oFeed->HTML;
}
__finally
{
  delete oFeed;
}
```

From then on, every new item is one **AddActivity** plus one push. The browsers already looking at the page grow the feed in place.

```
oFeed->AddActivity("Mark", "deleted", "draft #77", hcDanger, "bi bi-trash");
oEngine->BroadcastFragment(oFeed->GetLastItemFragmentHTML());
```

Activity Items

What it does: *Items* is a collection of **TsgcHTMLActivityItem**. You normally append to it with **AddActivity**, which stamps the timestamp and enforces the **MaxItems** cap, but the items can also be created at design time in the Object Inspector.

Item Property	Type	Default	Description
Id	string	''	Optional key of your own. When set, the element id of the row is built from it; when empty, the position in the collection is used instead.
Actor	string	''	Who did it. Rendered in bold at the start of the line.
ActorAvatarURL	string	''	Picture drawn in the round icon slot. When empty the Icon is used, and when that is empty too, the first letter of Actor.
Action	string	''	What they did ("updated", "deleted", "signed in").
Target	string	''	What it was done to ("invoice #1042"). Rendered in bold at the end of the line.
Timestamp	TDateTime	Now	When it happened. A new item is stamped with the current time, and AddActivity always re-stamps it.
Icon	string	''	A CSS icon class, for example `bi bi-pencil`, rendered as `<i class="...">` inside the icon circle.
Color	TsgcHTMLColor	hcPrimary	Background colour of the icon circle: hcPrimary, hcSecondary, hcSuccess, hcDanger, hcWarning, hcInfo, hcLight, hcDark, hcWhite or hcMuted.

The feed is rendered **newest first**: the last item added is drawn at the top. Actor, action and target are HTML-encoded, and the avatar URL is sanitized, so values coming from your data cannot inject markup.

Live Updates over WebSocket

What it does: this is the point of the component. The feed grows one item at a time, and one item is exactly what you send: not the page, not the list, one row.

The list container carries a stable element id (**FeedID** followed by `_list`). **GetLastItemFragmentHTML** renders the item you have just added as an htmx out-of-band fragment with an `afterbegin` swap on that container, which means: when this markup reaches the browser, insert it at the top of the list and leave everything else alone. That matches the newest-first order of the feed.

The fragment is sent through the WebSocket connection the page already holds, using the sgcHTML engine:

Engine Method	Description
`procedure PushFragment(const aGuid, aHTML: string);`	Sends the fragment to a single client, addressed by its connection guid.

`procedure BroadcastFragment(const aHTML: string; const aChannel: string = ''; const aExclude: string = '');`	Sends the fragment to every connected client, optionally only those on a channel, optionally excluding one guid.

The browser applies the swap in place. There is no HTTP round-trip, no polling and no client-side data binding: the server owns the feed and pushes only the new row.

```
void __fastcall TServerForm::InvoiceUpdated(const String aUser, const String aInvoice)
{
  oFeed->AddActivity(aUser, "updated", aInvoice, hcSuccess, "bi bi-pencil");
  oEngine->BroadcastFragment(oFeed->GetLastItemFragmentHTML());
}
```

Use **PushFragment** when only one browser should see the entry, and pass a channel to **BroadcastFragment** when the feed is per project, per tenant or per room rather than global. **GetItemFragmentHTML** does the same for any item by position, when you need to replay an older entry into a browser that has just connected.

The empty state travels with the item. The *EmptyText* placeholder is always part of the markup, as the last child of the list container, with the id **FeedID** followed by `_empty`; it is simply hidden while there is at least one entry. Every item fragment carries a *second* out-of-band swap that re-renders that placeholder with the visibility it should now have, so the first entry pushed into a feed that was served empty hides the *No activity yet* message on its way in, instead of appearing above a stale one. Nothing extra to call: the two fragments are in the single string the method returns. When the feed empties out, push **GetEmptyFragmentHTML** to bring the message back.

Relative Timestamps

What it does: with **ShowRelativeTime** True (the default), each row shows "just now", "5 min ago", "2 hours ago", "3 days ago", and the text stays correct as time passes without any further traffic from the server.

The component writes the age of the item into the markup once, when the row is rendered, and emits a small script that re-renders the "... ago" text on a timer in the browser. The script anchors each row against the client clock the first time it sees it, so no timezone conversion is needed, and it watches the document for new rows, which is why an item pushed over the WebSocket ages correctly from the moment it arrives, with no further server round-trip. The text is written with `textContent`, never as markup.

Set **ShowRelativeTime** to False to print the absolute timestamp instead, formatted with **TimeFormat**. No script is emitted then, and the timestamps are fixed.

Appearance

What it does: *Title*, *Compact* and *EmptyText* tune the frame around the items; the colour and the icon of each row come from the item itself.

Property	Type	Default	Description
Items	TsgcHTMLActivityItems	Empty collection	The feed: a collection of TsgcHTMLActivityItem items, rendered newest first.

FeedID	string	''	The id of the generated root element, and the prefix of the list container id that pushed fragments target. Leave it empty and a unique id is generated on the first render, so two feeds on one page never collide.
MaxItems	Integer	50	Ring-buffer cap. When AddActivity exceeds it, the oldest items are dropped. 0 means no cap.
ShowRelativeTime	Boolean	True	Shows "3 minutes ago" instead of an absolute timestamp, kept live in the browser.
Compact	Boolean	False	Tightens the vertical padding of each row.
Title	string	'Activity'	Heading drawn above the feed. Empty hides it.
EmptyText	string	'No activity yet'	Text drawn when the feed has no items.
TimeFormat	string	'yyyy-mm-dd hh:nn'	Format of the absolute timestamp, used when ShowRelativeTime is False.

Output

The inherited **HTML** property returns a `<div>` carrying the **FeedID**, an optional title, and an inner list `<div>` holding one element per item, newest first, followed by the empty-state placeholder (hidden while there are items). When **ShowRelativeTime** is True, an inline `<script>` is appended that keeps the "... ago" texts live in the browser. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 8 properties available on the component.
- Methods — 4 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_JobProgress

TsgcHTMLComponent_JobProgress monitors long-running server jobs, showing each one as a row with a progress bar and a status, and pushes the progress of a single job to the browser over the WebSocket the page already has open.

Introduction

TsgcHTMLComponent_JobProgress, declared in the unit **sgcHTML_Component_JobProgress**, renders a list of jobs: an import, a report build, a batch task, anything that runs on the server for long enough that the user wants to see how far it has got. Each job is a row with a name, an optional description, a percentage bar, a status badge and, while the job can still be stopped, a Cancel button.

The point of the component is the second part: you do not re-render the page on every progress tick. Each job row carries a stable element id, so you render only that row and send it to the browser as an out-of-band fragment over the WebSocket connection. The section *Live Progress over WebSocket* below explains the whole mechanism.

Quick Start

1. Drop a **TsgcHTMLComponent_JobProgress** component on the form (or create it at runtime) and assign the **HTMXEngine** that serves the page.

2. Add a job with **AddJob**, and report progress with **UpdateJob**.

3. Push the row of that single job to the connected browsers with **GetJobFragmentHTML**.

```
TsgcHTMLComponent_JobProgress *oJobs = new TsgcHTMLComponent_JobProgress(NULL);
try
{
  oJobs->HTMXEngine = oEngine;
  oJobs->AddJob("imp1", "Import customers");
  oJobs->UpdateJob("imp1", 40, jsRunning);
  oEngine->BroadcastFragment(oJobs->GetJobFragmentHTML("imp1"));
}
__finally
{
  delete oJobs;
}
```

Jobs and Status

What it does: *Jobs* is a collection of **TsgcHTMLJobItem** items, one per row, rendered in collection order. Add a job with **AddJob**, report progress with **UpdateJob** and drop it with **RemoveJob**.

Property	Type	Default	Description
Id	string	''	Identifies the job. Every method takes this id, and the element id of the row is derived from it.

Name	string	''	Shown in bold at the top of the row.
Description	string	''	Optional second line. Nothing is rendered when it is empty.
Percent	Integer	0	The width of the progress bar and the number shown under it. Clamped to 0..100 by UpdateJob.
Status	TsgcHTMLJobStatus	jsQueued	The state of the job. Colours the badge and the bar.
StatusText	string	''	Overrides the text of the badge. Empty uses the name of the status.
StartedAt	TDateTime	0	Set by AddJob when the job is first created.

TsgcHTMLJobStatus has five values, and each one colours the badge and the progress bar: `jsQueued` (grey), `jsRunning` (blue, with an animated striped bar), `jsCompleted` (green), `jsFailed` (red) and `jsCancelled` (amber). The badge normally reads Queued, Running, Completed, Failed or Cancelled; set *StatusText* on the job, or pass it to **UpdateJob**, to show your own text instead, for example `1.240 of 5.000 rows`.

Live Progress over WebSocket

What it does: updates one row of the list in every connected browser, without polling, without an HTTP round-trip and without re-rendering the page.

The mechanism is this:

1. Every job row is rendered with a stable element id built from **JobsID** and the job *Id*, in the form `<JobsID>_job_<Id>`. Characters outside `A-Z a-z 0-9 - _` are replaced with an underscore, so an id that comes from a queue or a database is always a valid element id.

2. On each progress tick your worker calls **UpdateJob** with the new percentage and status, and then renders *only that row* with **GetJobFragmentHTML**. The result is a small out-of-band fragment (an htmx `hx-swap-oob="outerHTML"` swap) that targets the element id of that row.

3. The fragment is sent to the browsers through the HTMX engine, with `TsgcHTMX_Engine_Server.PushFragment(const aGuid, aHTML: string)` to reach a single client, or `TsgcHTMX_Engine_Server.BroadcastFragment(const aHTML: string; const aChannel: string = ''; const aExclude: string = '')` to reach every client (optionally restricted to a channel, optionally excluding one connection). It travels over the WebSocket connection the page already has open, so there is no polling, no new request and nothing else on the page is touched: the browser replaces that one row.

A new job needs a different fragment. An `outerHTML` swap replaces an element that is already in the document, so it cannot create a row that the client has never seen. A job you have just added with **AddJob** must therefore be pushed with **GetJobAppendFragmentHTML**, which returns a `beforeend` fragment that appends the row to the list container `<JobsID>_list`. The pattern is: **AddJob** then push **GetJobAppendFragmentHTML**, and from then on **UpdateJob** then push **GetJobFragmentHTML**.

Removal uses the same call as an update. GetJobFragmentHTML returns a delete fragment, instead of the row, when the job is no longer there (because **RemoveJob** was called, or because **AutoRemoveCompleted** is True and the job has just completed) or when the job is hidden because **ShowCompleted** is False. So the host always does the same thing after a change, push what **GetJobFragmentHTML** returns, and the row is updated or removed accordingly.

The empty state is kept in step for you. The *EmptyText* placeholder is always part of the markup, as the first child of the list container, with the id *JobsID* + `_empty`; it is simply hidden while a job is on show. Both **GetJobAp-**

pendFragmentHTML and **GetJobFragmentHTML** carry a *second* out-of-band swap that re-renders that placeholder with the visibility it should now have, so an appended job hides the *No jobs running* message on its way in, and the push that deletes the last row brings it back. Nothing extra to call. When the host empties the *Jobs* collection without pushing a row, push **GetEmptyFragmentHTML** to re-render the placeholder on its own.

```
// 1. A new job: append its row to the list on every client
oJobs->AddJob("imp1", "Import customers", "customers.csv");
oEngine->BroadcastFragment(oJobs->GetJobAppendFragmentHTML("imp1"));

// 2. Progress: update that row only, tick after tick
for (int i = 0; i <= 100; i++)
{
  oJobs->UpdateJob("imp1", i, jsRunning,
    String().sprintf(L"%d of 100 rows", i));
  oEngine->BroadcastFragment(oJobs->GetJobFragmentHTML("imp1"));
  Sleep(200);
}

// 3. Done. If AutoRemoveCompleted is True this same call removes the row
oJobs->UpdateJob("imp1", 100, jsCompleted);
oEngine->BroadcastFragment(oJobs->GetJobFragmentHTML("imp1"));
```

Use **PushFragment** instead of **BroadcastFragment**, with the GUID of the connection, when the job belongs to one user and the other users must not see it.

Cancelling a Job

What it does: a Cancel button is rendered on the rows of the jobs that can still be stopped, that is, the jobs whose status is `jsQueued` or `jsRunning`, when **ShowCancelButton** is True. Completed, failed and cancelled jobs show no button.

The button is not a link and it does not post a form to the server. It is a `<form data-sgc-ws-send>`, the same mechanism the Grid and the Chat components use, so `window.sgcHTMX.send` takes the fields of the form and sends them back to the server over the WebSocket that is already open. The fields are the action name held in **CancelAction**, the *Id* of the job, and the id of the component. The page also dispatches a DOM CustomEvent named `sgcJobProgress:cancel` with the same values in its detail, so the page can react locally as well, for example to grey the row out at once.

Nothing is cancelled by the component itself. The message arrives in your server code, typically in the **OnHTMXMessage** event of **TsgcHTMX_Engine_Server**, where you compare the action with **CancelAction**, stop your worker, and then report the new state the same way as any other progress change: **UpdateJob** with the status `jsCancelled` followed by a push of **GetJobFragmentHTML**.

```
void __fastcall TForm1::OnHTMXMessage(TObject *Sender, const String aGuid,
  const String aMessage)
{
  // aMessage carries the fields of the cancel form: action, job, component
  if (GetAction(aMessage) == oJobs->CancelAction)
  {
    String vJob = GetJobId(aMessage);
    CancelWorker(vJob);
    oJobs->UpdateJob(vJob, 0, jsCancelled);
    oEngine->BroadcastFragment(oJobs->GetJobFragmentHTML(vJob));
  }
}
```

Appearance

What it does: *Title* draws a heading above the list, *EmptyText* replaces the list when no job is visible, *ShowCompleted* and *AutoRemoveCompleted* decide what happens to a job once it finishes, and *JobsID* is the id of the root element and the prefix of every id the component generates.

Property	Type	Default	Description
JobsID	string	''	Id of the root element. The list container is <JobsID>_list and each row is <JobsID>_job_<Id>. Leave it empty and a unique id is generated on the first render, so two job lists on one page never collide.
Title	string	'Jobs'	Heading drawn above the list. Empty hides the heading.
EmptyText	string	'No jobs running'	Text shown when no job is visible.
ShowCompleted	Boolean	True	When False, jobs with the status jsCompleted are hidden. Failed and cancelled jobs stay visible.
AutoRemoveCompleted	Boolean	False	When True, UpdateJob deletes the job as soon as it is reported as jsCompleted.
ShowCancelButton	Boolean	True	Shows the Cancel button on queued and running jobs.
CancelCaption	string	'Cancel'	Caption of the Cancel button.
CancelAction	string	'job:cancel'	Action name sent back to the server when the button is pressed.

Output

The inherited **HTML** property returns a `<div>` carrying the *JobsID* as its id, containing the optional title, the list container `<JobsID>_list` holding the empty-state placeholder (hidden while a job is on show) followed by one row per visible job, and the inline script that turns the Cancel form into a WebSocket message and dispatches the `sgcJobProgress:cancel` event. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

That full markup is what the browser loads once. Everything after it is a fragment: **GetJobAppendFragmentHTML** for a new row, **GetJobFragmentHTML** for an update or a removal.

Reference

- Properties — 9 properties available on the component.
- Methods — 6 public methods.

- Events — this component has no published events.

TsgcHTMLComponent_LogViewer

TsgcHTMLComponent_LogViewer renders a streaming log / terminal view whose lines are kept in a ring buffer, and pushes each new line to the browser over the WebSocket connection the page already has open.

Introduction

TsgcHTMLComponent_LogViewer emits a monospaced log panel: an optional toolbar (title, search box, level filter and an auto-scroll checkbox) and a scrollable body that holds one `<div>` per log line. Every line carries a time-stamp, a level, an optional source and the text itself.

Lines live in a **ring buffer**. **AddLine** appends a line and then drops the oldest lines until the collection fits **MaxLines**, so a long-running tail never grows without limit.

The component renders server-side, and the interesting part is what happens next: a new line does *not* re-render the page and is not polled for. **GetLastLineFragmentHTML** returns a small out-of-band fragment that appends that one line to the log body, and the host sends it to the browsers over the WebSocket connection they already have open.

Quick Start

1. Drop a **TsgcHTMLComponent_LogViewer** component on the form (or create it at runtime).

2. Set the **Theme**, add a line with **AddLine** and read the inherited **HTML** property to serve the initial page.

3. From then on, every time you add a line, send **GetLastLineFragmentHTML** to the connected browsers with the HTMX engine.

```
TsgcHTMLComponent_LogViewer *oLog = new TsgcHTMLComponent_LogViewer(NULL);
try
{
  oLog->Theme = ltDark;
  oLog->Title = "Server Log";
  oLog->AddLine(llInfo, "server started");
  html = oLog->HTML;
}
__finally
{
  delete oLog;
}
```

Once the page is on screen, a new line is a single call plus a single push:

```
oLog->AddLine(llError, "connection lost", "net");
oEngine->BroadcastFragment(oLog->GetLastLineFragmentHTML());
```

Log Lines and Levels

What it does: *Lines* is a collection of **TsgcHTMLLogLine** items. Each item has a *Timestamp* (set to `Now` when the item is created), a *Level* (`llInfo` by default), an optional *Source* and the *Text*. You normally do not touch the collection directly: call **AddLine**, which appends the item, stamps *Timestamp* with `Now` and then trims the collection to **MaxLines**. **Clear** removes every line.

The level drives the CSS class of the line, the label printed in the level column and the position of the line in the level filter.

Level	Label	Key	Description
llTrace	TRACE	trace	Lowest level. Very fine-grained tracing.
llDebug	DEBUG	debug	Diagnostic detail useful while developing.
llInfo	INFO	info	Normal operation. The default level of a new line.
llWarning	WARN	warning	Something unexpected that did not stop the operation.
llError	ERROR	error	An operation failed.
llFatal	FATAL	fatal	Highest level. Rendered with a highlighted background.

Ring buffer: MaxLines caps how many lines the component keeps. Every **AddLine** deletes lines from the front of the collection until `Lines.Count` is no greater than **MaxLines**, so the oldest line is the one that goes. Set **MaxLines** to 0 (or a negative value) to disable trimming and keep every line.

Live Streaming over WebSocket

What it does: The component renders server-side. When a new line arrives you do not re-render the page and the browser does not poll for it. You render just that one line and push it down the WebSocket connection the page already has open.

GetLineFragmentHTML(aIndex) returns an out-of-band swap fragment (htmx `hx-swap-oob` with `beforeend`) that *appends* the line at `aIndex` to the log body. The fragment targets the body element, whose id is *LogID* + `_body`. If `aIndex` is out of range the function returns an empty string.

GetLastLineFragmentHTML is the one you normally use. It is exactly `GetLineFragmentHTML(Lines.Count - 1)`, that is, the line you just added with **AddLine**.

The empty state travels with the line. The *EmptyText* placeholder is always part of the markup, as the first child of the body, with the id *LogID* + `_empty`; it is simply hidden while there is at least one line. Every line fragment carries a *second* out-of-band swap that re-renders that placeholder with the visibility it should now have, so a line pushed into a viewer that was served empty hides the *No log entries* message on its way in, instead of landing underneath it. Nothing extra to call: the two fragments are in the single string the method returns. After **Clear**, push **GetEmptyFragmentHTML** to bring the message back.

The host sends the fragment with the HTMX engine server:

Method	Description
procedure PushFragment(const aGuid, aHTML: string);	Sends the fragment to a single client, addressed by its connection guid.

procedure BroadcastFragment(const aHTML: string; const aChannel: string = ''; const aExclude: string = '');	Sends the fragment to every connected client, optionally restricted to a channel and optionally excluding one connection guid.

The browser appends the line to the log body, the component's script reapplies the current level filter and search text, and, when *AutoScroll* is set, the body scrolls to the newest line. There is no polling and no HTTP round-trip.

```
// one client, by connection guid
oLog->AddLine(llWarning, "disk usage above 90%", "storage");
oEngine->PushFragment(vGuid, oLog->GetLastLineFragmentHTML());

// every client watching the 'logs' channel
oLog->AddLine(llError, "connection lost", "net");
oEngine->BroadcastFragment(oLog->GetLastLineFragmentHTML(), "logs");
```

Toolbar: Filter, Search and Autoscroll

What it does: The toolbar sits above the log body and holds the *Title*, the search box, the level filter and the auto-scroll checkbox. It is omitted entirely when *Title* is empty, *ShowSearch* and *ShowLevelFilter* are False, and no auto-scroll checkbox is rendered (that is, *AutoScroll* or *ShowAutoScrollToggle* is False).

Level filter (*ShowLevelFilter*): a drop-down list with an *All levels* entry plus one entry per level, labelled `TRACE+`, `DEBUG+`, `INFO+`, `WARN+`, `ERROR+` and `FATAL+`. It is a *minimum* level filter: picking `WARN+` shows warnings, errors and fatals and hides the rest. The filter runs in the browser and only changes the CSS display of the lines already rendered. Nothing is requested from the server.

Search (*ShowSearch*): a case-insensitive substring match against the visible text of each line (timestamp, level, source and text). Non-matching lines are hidden, matching lines are highlighted. Like the filter, it runs in the browser and matches only the lines currently in the body. *SearchPlaceholder* is the placeholder text of the box.

Auto-scroll (*AutoScroll*): decides whether the pane *follows* the newest line. When it is True the body scrolls down to the tail whenever a line is appended, and it also tracks the scroll position: if the reader scrolls up and away from the bottom, new lines stop dragging the view down, and scrolling back to the bottom resumes following. When it is False no scrolling code is emitted at all, so the pane stays exactly where the reader left it, whatever arrives over the WebSocket.

Auto-scroll checkbox (*ShowAutoScrollToggle*): the checked *Auto-scroll* checkbox at the right of the toolbar, which lets the reader pause the following at runtime. It is rendered when, and only when, *AutoScroll* and *ShowAutoScrollToggle* are both True: with *AutoScroll* False there is no following, so there is nothing to pause and no checkbox. Clear *ShowAutoScrollToggle* for a viewer that must always follow the tail with no switch offered to the reader, for example a wall-board.

A `MutationObserver` watches the body, so a line pushed over WebSocket is filtered, and scrolled into view when the viewer is following, exactly like a line rendered with the page.

Security

Log text, source and level labels are treated as **untrusted** data. Every one of them is emitted through the escaping node layer (`sgcHTMLEncode`), and values written into attributes go through `sgcHTMLAttrEncode`. The inline script only ever reads with `textContent` and writes with `textContent`, `classList` and `style`, never with `innerHTML`.

The practical consequence: nothing that arrives in a log line, wherever it came from, can turn into markup or script in the page. You can log raw request bodies, exception messages or remote input without sanitising them first.

Appearance

What it does: *Theme* picks the dark or light colour scheme, *Height* sets the height of the scrolling body, *WrapLines* decides whether a long line wraps or scrolls horizontally, and *ShowTimestamps* plus *TimeFormat* control the time column.

Property	Type	Default	Description
Lines	TsgcHTMLLogLines	Empty collection	The log lines currently held by the component.
LogID	string	''	The id of the generated root element. The body, the empty-state placeholder, the search box, the level filter and the auto-scroll checkbox derive their ids from it. Empty by default: a unique id is generated on the first render.
MaxLines	Integer	500	Ring buffer size. AddLine drops the oldest lines beyond this count.
AutoScroll	Boolean	True	Decides whether the log body follows the newest line. False emits no scrolling code at all.
ShowAutoScrollToggle	Boolean	True	Renders the Auto-scroll checkbox so the reader can pause the following. Only meaningful while AutoScroll is True.
ShowTimestamps	Boolean	True	Shows the timestamp column.
ShowLevelFilter	Boolean	True	Shows the minimum-level drop-down list in the toolbar.
ShowSearch	Boolean	True	Shows the search box in the toolbar.
Height	string	'400px'	Height of the scrolling log body, as a CSS length.
WrapLines	Boolean	False	Wraps long lines instead of scrolling them horizontally.
Theme	TsgcHTMLLogTheme	ltDark	Colour scheme: ltDark or ltLight.
Title	string	''	Optional caption shown at the left of the toolbar.
EmptyText	string	'No log entries'	Text shown in the body while there are no lines.
SearchPlaceholder	string	'Search...'	Placeholder text of the search box.
TimeFormat	string	'hh:nn:ss'	FormatDateTime pattern used for the timestamp column.

Output

The inherited **HTML** property returns a `<div>` carrying the *LogID*, the toolbar markup, a scrolling body `<div>` holding the empty-state placeholder followed by one line per entry, and an inline script that wires up the level filter, the search box, the auto-scroll checkbox (when one is rendered) and the `MutationObserver` that handles pushed lines. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 15 properties available on the component.
- Methods — 5 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_ChatBox

TsgcHTMLComponent_ChatBox renders a compact, Bootstrap-styled chat panel: a scrolling list of message bubbles with avatars and timestamps, an optional typing indicator and an optional message input box.

Introduction

TsgcHTMLComponent_ChatBox emits the HTML for a self-contained chat card. You add messages to its **Messages** collection (or with the **AddMessage** helper); each message has a sender, text, timestamp, an alignment (left or right) and a Bootstrap contextual color. The component builds the bubble markup, the avatar initials, the scroll-to-bottom script and an optional input form, then returns the complete markup through its inherited **HTML** property.

The panel is built on Bootstrap utility classes, so it inherits the look of whatever Bootstrap theme the surrounding page uses. No Content Delivery Network is required at runtime; the sgcHTML engine serves the styles and scripts it needs.

Quick Start

1. Drop a **TsgcHTMLComponent_ChatBox** component on the form (or create it at runtime).

2. Set the **Title** and add a couple of messages with **AddMessage**, choosing the alignment and color.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_ChatBox *oChat = new TsgcHTMLComponent_ChatBox(NULL);
try
{
  oChat->Title = "Support";
  oChat->AddMessage("Agent", "Hello, how can I help?", caLeft, hcSecondary);
  oChat->AddMessage("You", "My order has not arrived.", caRight, hcPrimary);
  html = oChat->HTML;
}
__finally
{
  delete oChat;
}
```

Messages

What it does: *Messages* is a collection of **TsgcHTMLChatMessage** items. Each item exposes *Sender*, *Text*, *Timestamp*, *Align* (caLeft / caRight), *Color*, *AvatarInitials* and an optional *ElementID*. You can populate the collection in the Object Inspector, or call **AddMessage** at runtime, which fills the timestamp and avatar initials for you.

Property	Type	Default	Description
Messages	TsgcHTMLChatMessages	–	Collection of TsgcHTMLChatMessage items rendered as bubbles.

Title	string	'Chat'	Caption drawn in the panel header. Leave empty to hide the header.
Height	string	'400px'	CSS height of the scrolling message area.
ChatID	string	'sgc-Chat'	The id of the generated card and the prefix for the messages, typing and input element ids.

```
oChat->Messages->Clear();
TsgcHTMLChatMessage *oMsg = oChat->Messages->Add();
oMsg->Sender = "Agent";
oMsg->Text = "Welcome!";
oMsg->Align = caLeft;
oMsg->Color = hcInfo;
```

Input and Typing Indicator

What it does: *ShowInput* draws a message form at the foot of the panel; *InputPlaceholder*, *SendButtonText* and *SendButtonStyle* configure that form; the placeholder is attribute-encoded when it is written. *ShowTypingIndicator* shows an animated "typing…" row, and *TypingText* sets its caption. The input form posts through the sgcHTML WebSocket bridge so a server handler can append the reply.

Property	Type	Default	Description
ShowInput	Boolean	True	Shows or hides the message input form at the foot of the panel.
InputPlaceholder	string	'Type a message...'	Placeholder text shown in the empty input field.
SendButtonText	string	'Send'	Caption of the send button.
SendButtonStyle	TsgcHTMLButtonStyle	bsPrimary	Bootstrap button style applied to the send button.
ShowTypingIndicator	Boolean	False	Shows the animated typing-indicator row.
TypingText	string	'typing...'	Caption shown next to the typing dots.

Output

The inherited **HTML** property returns a Bootstrap `<div class="card">` containing the optional header, the scrolling messages area, the optional typing row and the optional input form, plus an inline script that keeps the panel scrolled to the newest message. Serve this string directly, or assign it to the BodyContent of a

TsgcHTMLTemplate_Bootstrap to produce a full HTML document. When streaming new messages over a WebSocket, send only the fragment returned by **GetLastMessageHTML**.

Reference

- Properties — 10 properties available on the component.
- Methods — 2 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_Chat

TsgcHTMLComponent_Chat renders a rich, messenger-style conversation: a coloured header with avatar, grouped message bubbles with read receipts, replies, forwarded marks, date separators and image, file, audio and video attachments.

Introduction

TsgcHTMLComponent_Chat emits the HTML for a full conversation surface styled after a modern mobile messenger. You add messages to its **Messages** collection, or with the **AddMessage**, **AddImageMessage**, **AddFileMessage** and **AddSystemMessage** helpers. Each message has a sender, text or media, a timestamp, an alignment, a delivery status and optional reply and forwarded marks. The component groups consecutive messages from the same sender, draws date separators, renders read receipts and returns the complete markup through its inherited **HTML** property.

The look is tuned with the **Options** sub-object: bubble colors, background, maximum bubble width and which decorations to show. The component carries its own CSS, so no Content Delivery Network is required at runtime.

Quick Start

1. Drop a **TsgcHTMLComponent_Chat** component on the form (or create it at runtime).

2. Set the **Title** and **Subtitle**, then add a few messages with **AddMessage**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_Chat *oChat = new TsgcHTMLComponent_Chat(NULL);
try
{
  oChat->Title = "Alice";
  oChat->Subtitle = "online";
  oChat->AddMessage("Alice", "Hey, are you there?", maLeft);
  oChat->AddMessage("You", "Yes, what is up?", maRight);
  html = oChat->HTML;
}
__finally
{
  delete oChat;
}
```

Header and Layout

What it does: The header shows the conversation *Title*, a *Subtitle* (an “online” subtitle draws a green presence dot) and an avatar from *HeaderAvatarURL* or *HeaderAvatarInitials*. *Height* sets the overall panel height and *ChatID* the element id.

Property	Type	Default	Description
Title	string	‘Chat’	Conversation name drawn in the header.

Subtitle	string	''	Secondary header line. A subtitle containing "online" shows a presence dot.
HeaderAvatarURL	string	''	Image URL for the header avatar.
HeaderAvatarInitials	string	''	Initials shown when no avatar URL is set.
Height	string	'500px'	Overall height of the chat panel.
ChatID	string	'sgcChat'	The id of the generated container and prefix for inner element ids.

Messages and Media

What it does: *Messages* is a collection of **TsgcHTMLChat_Message** items. Beyond plain text a message can carry an image, file, audio or video attachment, a reply quote, a forwarded mark and a delivery status. Add a plain message with **AddMessage**, an image with **AddImageMessage**, a file with **AddFileMessage** and a centred notice with **AddSystemMessage**.

```
oChat->AddImageMessage("Alice", "/img/photo.jpg", maLeft, hcPrimary, "Look at this!");
oChat->AddFileMessage("You", "/files/report.pdf", "report.pdf", "256 KB", maRight);
oChat->AddSystemMessage("Alice joined the chat");
```

Options

What it does: The *Options* sub-object (**TsgcHTMLChat_Options**) tunes the appearance and which decorations are drawn: *MaxBubbleWidth*, *SentBubbleColor*, *ReceivedBubbleColor*, *BackgroundColor*, *BackgroundPattern*, *ShowTimestamp*, *ShowStatus*, *ShowAvatar*, *GroupConsecutive* and *ShowDateSeparators*.

```
oChat->Options->SentBubbleColor = "#dcf8c6";
oChat->Options->ShowStatus = true;
oChat->Options->GroupConsecutive = true;
```

Input and Events

What it does: When *ShowInput* is True the panel draws a messenger input bar with optional attach (*ShowAttachButton*) and emoji (*ShowEmojiButton*) buttons; *InputPlaceholder* sets the hint text. *ShowTypingIndicator* and *TypingText* control the typing animation. The component publishes **OnSendMessage**, raised when the user submits a message, and **OnMediaClick**, raised when the user clicks an attachment.

Output

The inherited **HTML** property returns the chat container with its header, scrolling messages area, optional typing row and optional input bar, plus an inline script that keeps the newest message in view. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap**. When a single new message arrives, push the fragment returned by **GetLastMessageHTML** instead of re-rendering the whole conversation.

Reference

- Properties — 14 properties available on the component.
- Methods — 5 public methods.
- Events — 2 published events.

TsgcHTMLComponent_AIChat

TsgcHTMLComponent_AIChat is an AI-assistant chat panel: a model-selector header, a welcome message, token-by-token streaming, a conversation-history JSON helper and optional RAG source attribution. It descends from TsgcHTMLComponent_ChatBox.

Introduction

TsgcHTMLComponent_AIChat extends TsgcHTMLComponent_ChatBox with everything needed to front a Large Language Model: a header that shows the provider and model, an opening **WelcomeMessage**, distinct colors for the user and the assistant, streaming helpers that append a reply chunk by chunk, a **GetConversationHistoryJSON** method that builds the message array you send to the API, and Retrieval-Augmented Generation (RAG) support that attaches source references to an answer.

Because it inherits from TsgcHTMLComponent_ChatBox, the **Messages** collection, **AddMessage**, **GetLastMessageHTML** and the input and typing-indicator properties are all available too. The high-level flow is: the user submits a message, you call **ProcessUserMessage**, that raises **OnRAGContext** (when RAG is enabled) and then **OnChatSend**; you call the model and stream the answer back with **BeginStreaming**, **PushStreamChunk** and **EndStreaming**.

Quick Start

1. Drop a **TsgcHTMLComponent_AIChat** component on the form (or create it at runtime).

2. Set the **AIProvider**, **ModelName** and **WelcomeMessage**, and handle **OnChatSend** to call your model.

3. Read the **HTML** property and serve it; stream the answer back with the streaming methods.

```
TsgcHTMLComponent_AIChat *oAI = new TsgcHTMLComponent_AIChat(NULL);
try
{
  oAI->AIProvider = apOpenAI;
  oAI->ModelName = "gpt-4o";
  oAI->WelcomeMessage = "Hi! Ask me anything.";
  oAI->OnChatSend = DoChatSend;
  html = oAI->HTML;
}
__finally
{
  delete oAI;
}
```

Provider and Appearance

What it does: *AIProvider* and *ModelName* drive the header pill that shows which model is in use; *ShowModelSelector* toggles that header. *AIName*, *UserColor* and *AIColor* style the two speakers, *WelcomeMessage* is shown when the conversation is empty and *SystemPrompt* holds the instruction you pass to the model. *AIChatID* sets the element id.

Property	Type	Default	Description
AIProvider	TsgcHTM-LAIProvider	apOpe-nAI	Provider shown in the header: apOpenAI, apAn-thropic, apGemini or apCustom.
ModelName	string	'gpt-4o'	Model name shown in the header pill.
ShowModelSe-lector	Boolean	True	Shows or hides the provider/model header.
SystemPrompt	string	''	System instruction passed to the model. Not ren-dered in the panel.
AIName	string	'AI Assis-tant'	Sender name used for assistant messages.
WelcomeMes-sage	string	(greeting)	Opening assistant message shown when the con-versation is empty.
UserColor	TsgcHTMLColor	hcPrima-ry	Bubble color for the user's messages.
AIColor	TsgcHTMLColor	hcSuc-cess	Bubble color for the assistant's messages.
AIChatID	string	'sg-cAIChat'	The id of the generated panel and prefix for inner ids.

Streaming

What it does: When *StreamingEnabled* is True you can append the assistant's reply progressively. Call **Begin-Streaming** to open an empty assistant bubble, **PushStreamChunk** for each token or fragment you receive from the model, and **EndStreaming** when the reply is complete. **GetStreamFragmentHTML** returns the htmx fragment that updates the open bubble in place.

```
oAI->BeginStreaming();
oAI->PushStreamChunk("The ");
oAI->PushStreamChunk("answer ");
oAI->PushStreamChunk("is 42.");
oAI->EndStreaming();
```

RAG (Retrieval-Augmented Generation)

What it does: With *RAGEnabled* set, **ProcessUserMessage** raises the **OnRAGContext** event so you can return retrieved context and source references for the query before the model is called. The answer is then attached with **AddAIMessageWithSources**, which renders the sources according to *RAGDisplayMode* (rdInline, rdCollapsible or rdFootnotes) under the *RAGLabel* heading. *ShowSourceReferences* turns the attribution block on or off.

Property	Type	Default	Description
RAGEnabled	Boolean	False	Enables the RAG context lookup before the model is called.
RAGDisplayMode	TsgcRAGDisplayMode	rdCollapsible	How sources are shown: rdInline, rdCollapsible or rdFootnotes.
RAGLabel	string	'Sources'	Heading for the source-references block.
ShowSourceReferences	Boolean	True	Shows or hides the source-references block.
StreamingEnabled	Boolean	True	Enables progressive (chunk-by-chunk) streaming of the answer.

Output

The inherited **HTML** property returns the assistant panel: the model-selector header, the scrolling messages area (seeded with the welcome message), an optional "thinking…" typing row and the input form, plus the scroll script. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap**. During a reply, push the fragment from **GetStreamFragmentHTML** to grow the open bubble in place.

Reference

- Properties — 14 properties added by this component (plus the inherited ChatBox properties).
- Methods — 7 public methods for streaming, RAG and history.
- Events — 2 published events.

TsgcHTMLAIAssistant

Turns a question in plain language into a validated set of operations over your own data.

Introduction

TsgcHTMLAIAssistant is the non visual component of the AI Data Pack. It turns a question typed in plain language into a **validated operation set**: a closed list of filters, sorts, a grouping, highlights and a limit for a grid, or rows, columns, a measure and an aggregate for a pivot. It never touches your dataset, never builds SQL and never emits HTML. See AI Data Pack for the whole picture.

Assign it to the **AIAssistant** property of TsgcHTMLComponent_Grid, TsgcHTMLComponent_DataTable, TsgcHTMLComponent_PivotTable, TsgcHTMLComponent_Form or TsgcHTMLComponent_InlineAIPrompt and those components do the rest. You can also call it directly, which is what the methods below are for.

A provider is optional. With none configured, and on every platform where the AI units are not compiled in, a rule-based parser answers the grid, the pivot and the smart paste calls, so a page works with no API key at all.

The component lives in the unit **sgcHTML_AI** and is registered on the **SGC HTML** palette page.

Quick Start

1. Drop a **TsgcHTMLAIAssistant** on the form, or create it at runtime.

2. Optionally configure a provider with **Provider**, **APIKey** and **Model**, or assign your own **Chat** component. Without one the rule-based parser answers.

3. Assign it to the **AIAssistant** property of the data component and set **AIQueryEnabled** to render the query bar.

4. Route the actions the controls post back into the component, as described in AI Data Pack.

```
FAI = new TsgcHTMLAIAssistant(this);
FAI->Provider = aiprOpenAI;
FAI->APIKey = "API_KEY";
FAI->Model = "gpt-4o-mini";
FAI->AllowRowDataToProvider = false;

oGrid->AIAssistant = FAI;
oGrid->AIQueryEnabled = true;
```

The Provider

What it does: either the assistant owns a chat component configured from these properties, or it asks the one you assign to *Chat*. With neither, HasProvider is False and every answer comes from the rule-based parser.

Proper-ty	Type	Default	Description

Chat	TsgcAIChat	nil	Your own chat component. Only declared where the AI units are compiled in.
Provider	TsgcHTM-LAIProvider	aiprOpe-nAI	aiprOpenAI, aiprAnthropic, aiprGemini, aiprDeepSeek, aiprOllama, aiprGrok, aiprMistral.
APIKey	string	''	Credential of the owned chat component.
Model	string	''	Model name. Empty leaves the default of the chat compo-nent.
BaseUrl	string	''	Endpoint base, enough on its own to count as a provider.

Each call asks with structured output and a JSON schema, and restores your chat component to the state it was found in.

What Travels, and What Does Not

What it does: *AllowRowDataToProvider* is the row-data gate, and it is False by default.

Property	Type	De-fault	Description
AllowRow-DataToProvider	Boolean	False	False keeps every row inside the process: SuggestColumn-Values and Summarize return False and send nothing.
MaxRows	Integer	200	The most rows a call that sends rows may carry.
MaxLimit	Integer	1000	Ceiling every limit the model returns is clamped to. It cannot be raised over 1000.

A grid or pivot question sends the column names, the captions, the types and up to three sample values per col-umn, whatever the gate says. That is the whole payload.

The Answer

What it does: these properties decide the language, the caching and what happens when the provider does not answer.

Property	Type	De-fault	Description
Language	string	''	Language the free text is asked in. Empty uses the language of the request locale.

CacheMinutes	Integer	5	Minutes an identical question over an identical column set is answered from the cache. 0 switches it off.
UseRuleBasedFallback	Boolean	True	Lets the rule-based parser answer a failed provider call. With no provider it always answers.

Events

Event	Fired
OnBeforeQuery	Before every provider call, with the whole prompt. Set aAllow to False to block it, which also skips the rule-based parser for that question.
OnAIError	When the provider call raised. No method raises: the exception is reported here and treated as no answer.

See Also

- AI Data Pack
- TsgcHTMLGridOps
- TsgcHTMLPivotOps
- TsgcHTMLAIColumns
- TsgcHTMLComponent_Grid
- TsgcHTMLComponent_InlineAIPrompt

Reference

- Properties: 11 properties available on the component.
- Methods: 8 public methods.
- Events: 2 events you can handle.

TsgcHTMLComponent_InlineAIPrompt

A popover that asks the assistant next to the control a human is working on, with Accept and Discard.

Introduction

TsgcHTMLComponent_InlineAIPrompt is a small popover that asks the assistant something right where the user is working: next to a memo, next to a rich editor or inside the header of a grid column. It is a human in the loop control and nothing else. The prompt is typed in the popover, the answer is shown in it, and only **Accept** reports the answer to the host, through AI Data Pack.

The component never writes anything into a dataset, into a field or into the page. Like every other sgcHTML component it renders markup, and the host decides what to do with what comes back. Model text is written through the node layer, so it is escaped and can never become markup.

It lives in the unit **sgcHTML_Component_InlineAIPrompt** and needs a TsgcHTMLAIAssistant to answer anything by itself.

Quick Start

1. Create the component, give it a **PromptID** and a **TargetName**, and assign the **AIAssistant**.

2. Fill **Context** with the rows the assistant should see, or handle **OnAIPrompt** and answer yourself.

3. Route the four actions the popover posts (inlineAIOpen, inlineAIAsk, inlineAIAccept, inlineAIDiscard) into Open, ProcessPrompt, Accept and Discard, and push the re-rendered component back.

4. Write the accepted value where it belongs in **OnAIAccept**.

```
FPrompt = new TsgcHTMLComponent_InlineAIPrompt(this);
FPrompt->PromptID = "notePrompt";
FPrompt->TargetName = "Note";
FPrompt->AIAssistant = FAI;
FPrompt->OnAIAccept = PromptAccept;
FPrompt->Context->Add("Customer=Northwind");

if (vAction == "inlineAIAsk")
  FPrompt->ProcessPrompt(vText);
```

Properties

Property	Type	Default	Description
PromptID	string	'sgcInlineAI'	Id of the popover, and the value posted in the prompt field.

Target-Name	string	''	Name of the control the answer belongs to, posted in the target field.
Title	string	''	Title of the panel. Empty uses the localized Ask text.
Placehold-er	string	''	Placeholder of the prompt box.
ButtonText	string	''	Caption of the trigger and of the ask button. Empty uses the localized Ask text.
IsOpen	Boolean	False	True renders the panel, False only the trigger.
AIAssistant	TsgcHTM-LAIAssis-tant	nil	The assistant asked when OnAIPrompt did not answer.
Context	TStringList	(empty)	Public. The rows the assistant sees, one line per row. Row data.
Prompt-Text	string	''	Public. What was asked.
ResultText	string	''	Public. What was answered. Escaped when it is rendered.

The Actions It Posts

There is no HTTP route. Every control is a `data-sgc-ws-send` form, and the host dispatches on the `action` field, checking the `prompt` field to know which popover sent it.

Action	Fields	Method to call
inlineAIOpen	prompt, target	Open
inlineAIAsk	prompt, target, text	ProcessPrompt(text)
inlineAIAccept	prompt, target	Accept
inlineAIDiscard	prompt, target	Discard

The `text` field carries what was typed in the prompt box, line breaks included.

Output

The trigger is always rendered. With *IsOpen* True a card is rendered next to it with the title, the prompt box, the ask button and, once there is an answer, the answer itself with an Accept and a Discard control. The texts come from the localized strings AIAsk, AIAccept and AIDiscard, see Localization.

See Also

- AI Data Pack
- TsgcHTMLAIAssistant
- TsgcHTMLComponent_Grid
- TsgcHTMLComponent_Form

Reference

- Properties: 10 properties available on the component.
- Methods: 4 public methods.
- Events: 2 events you can handle.

TsgcHTMLAIColumns

The column whitelist a question is validated against, and the only thing about your data it sends.

Introduction

TsgcHTMLAIColumns is the column whitelist: what the model is allowed to name, and the type every value it returns for a column is coerced by. It is the security boundary of the whole pack, and it is also the only thing about your data a grid or pivot question ever sends.

A data component builds one for you with its own GetAIColumns, and the caller frees it. Build one yourself when you call TsgcHTMLAIAssistant directly.

Columns

Each item is a **TsgcHTMLAIColumn**:

Property	Type	Default	Description
Name	string	''	The name the model may use, and the name the operations come back with.
Caption	string	''	What a human calls the column. The model is shown both, and an answer that names the caption resolves to the column.
DataType	TsgcHTMLAIDataType	aidtString	aidtString, aidtInteger, aidtFloat, aidtDate, aidtDateTime or aidtBoolean. Every value the model returns for this column is coerced by it, and a value that does not coerce drops its whole operation.
SampleCount	Integer	0	Read-only. How many sample values the column carries.
Samples[i]	string	(none)	Read-only. At most three samples, each capped at 40 characters. Empty and repeated values are ignored.

Methods

- **Add(aName, aDataType, aCaption)**: appends a column and returns it.
- **AddSample(aValue)** on a column: adds a sample value, up to three.
- **Find(aName)** and **IndexOf(aName)**: exact match, without case, against the Name first and then the Caption.
- **Fingerprint**: the name and the type of every column, which is the part of the cache key that invalidates a cached answer when the column set changes.
- **Clear**.

```
TsgcHTMLAIColumns *oColumns = new TsgcHTMLAIColumns();
try
{
  oColumns->Add("Amount", aidtFloat, "Total")->AddSample("1250.40");
  oColumns->Add("Country", aidtString, "Country")->AddSample("Spain");
  oAI->QueryToGridOps(vQuestion, oColumns, oOps);
}
__finally
{
  delete oColumns;
}
```

See Also

- AI Data Pack
- TsgcHTMLAIAssistant
- TsgcHTMLGridOps
- TsgcHTMLPivotOps

TsgcHTMLGridOps

The validated filters, sorts, grouping, highlights and limit a grid question produced.

Introduction

TsgcHTMLGridOps is the validated operation set of a grid question: what survived the column whitelist, the closed operator set and the type coercion. It is what TsgcHTMLAIAssistant produces and what a data component applies to its own rows. It carries no SQL and no expression, only names, operators and coerced values.

It round trips through JSON, so it can live in a request session or in a hidden field of the page, and it has a stable text form for the interpreted as chip. It is declared in the unit **sgcHTML_AI**.

Properties

Property	Type	Default	Description
Filters	TsgcHTMLAIFilters	(empty)	Filters, each with Column, Op, Value and Value2 (the upper bound of a between).
Sorts	TsgcHTMLAISorts	(empty)	Sorts, each with Column and Descending, applied in order.
Highlights	TsgcHTMLAIHighlights	(empty)	Highlights, each with Column, Op, Value and Style. The style is one of primary, secondary, success, danger, warning, info, light and dark.
GroupBy	string	''	Name of the column the rows are grouped by.
Limit	Integer	0	The most rows to show. 0 means no limit. It is clamped to MaxLimit.
Explanation	string	''	The free text the model wrote, sanitized. It is what the interpreted as chip shows.

The operators are aiopEquals, aiopNotEquals, aiopContains, aiopStartsWith, aiopEndsWith, aiopGreater, aiopGreaterOrEqual, aiopLess, aiopLessOrEqual, aiopBetween, aiopIsEmpty, aiopIsNotEmpty and aiopInList. Anything else in an answer is discarded.

Methods

- **ToJSON**: the operation set as a JSON document, the form a component keeps between two requests.

- **FromJSON(aJSON, aColumns, aMaxLimit)**: reads it back. **With aColumns assigned this is the full boundary**: every column name is resolved against the whitelist, every operator and style checked, every value coerced and the limit clamped. Called with aColumns nil it only checks the structure, which is not enough for text that came back from a browser.
- **ToText**: the stable text form, used for the chip when the answer carried no explanation.
- **Clear**, **IsEmpty** and **Assign**.

An operation set where nothing survived validation is cleared completely, the Explanation included, so a rejected answer leaves no model text behind.

See Also

- AI Data Pack
- TsgcHTMLAIAssistant
- TsgcHTMLPivotOps
- TsgcHTMLAIColumns

TsgcHTMLPivotOps

The validated rows, columns, measure and aggregate a pivot question produced.

Introduction

TsgcHTMLPivotOps is the validated layout of a pivot question: the row fields, the column fields, one measure and one aggregate. It is what TsgcHTMLAIAssistant produces for a pivot and what TsgcHTMLComponent_PivotTable lays itself out from.

Like the grid operation set it round trips through JSON and has a stable text form. It is declared in the unit **sgcHTML_AI**.

Properties

Property	Type	Default	Description
Rows	TStringList	(empty)	Names of the row fields, in order.
Columns	TStringList	(empty)	Names of the column fields, in order.
Measure	string	''	Name of the field the aggregate is computed over.
Aggregate	TsgcHTMLAIAggregate	aiagSum	aiagSum, aiagCount, aiagAvg, aiagMin or aiagMax.
Explanation	string	''	The free text the model wrote, sanitized.

Methods

- **ToJSON** and **FromJSON(aJSON, aColumns)**, with the same rule as the grid operation set: pass the live whitelist and every field name is resolved against it.
- **ToText**, **Clear**, **IsEmpty** and **Assign**.

A field name that is not in the whitelist is dropped, an aggregate that is not one of the five falls back to the sum, and a layout where nothing survived is cleared completely.

See Also

- AI Data Pack
- TsgcHTMLAIAssistant
- TsgcHTMLGridOps
- TsgcHTMLAIColumns

TsgcHTMLAuth

TsgcHTMLAuth adds cookie sessions, remember-me, CSRF tokens and login throttling to sgcHTML applications, on top of a session store.

Introduction

TsgcHTMLAuth is the authentication component of sgcHTML. It keeps the signed-in users in a session store (TsgcHTMLSessionStore_Memory or TsgcHTMLSessionStore_File), recognizes them by a session cookie, can keep them signed in for days with a remember-me cookie, protects state-changing requests with a CSRF token per session and throttles password guessing per client IP and per user name. Checking the password is up to you, in the **OnAuthenticate** event.

The component is transport-neutral: requests come in as strings (the Cookie header, the client IP, the User-Agent) and the cookies to send come out as complete Set-Cookie header values. Assign it to the **Auth** property of TsgcHTMX_Engine_Server, of TsgcHTMX_Engine_Server_WebBroker or of the http.sys engine, and the engine does the rest: it reads the session of every request, protects the routes marked with RequireLogin or RequireRoles, checks the CSRF token and serves the login and logout endpoints. You can also call it from your own HTTP handlers. Sessions, CSRF and Authentication explains the whole flow.

The component lives in the unit **sgcHTML_Auth** and is registered on the **SGC HTML** palette page, next to the two session stores.

Quick Start

1. Drop a **TsgcHTMLSessionStore_Memory** and a **TsgcHTMLAuth** (or create them at runtime) and assign the store to **SessionStore**.

2. Handle **OnAuthenticate** to check the user name and the password.

3. Assign the TsgcHTMLAuth to the **Auth** property of the engine and set **RequireLogin** on the routes to protect.

```
FStore = new TsgcHTMLSessionStore_Memory(this);
FStore->SweepInterval = 60;

FAuth = new TsgcHTMLAuth(this);
FAuth->SessionStore = FStore;
FAuth->OnAuthenticate = AuthAuthenticate;
FHTMX->Auth = FAuth; // TsgcHTMX_Engine_Server

TsgcHTMX_Route *oRoute = FRouter->Routes->Add();
oRoute->Path = "/orders";
oRoute->RequireLogin = true;
oRoute->OnRoute = OrdersRoute;
```

Session Store

What it does: *SessionStore* keeps the sessions and the remember-me tokens. It must be assigned before the first request.

Property	Type	Default	Description
SessionStore	TsgcHTMLSessionStore	nil	A TsgcHTMLSessionStore_Memory, a TsgcHTMLSessionStore_File or your own descendant.

Session Cookie

What it does: these properties name the cookies and set their attributes. The defaults are the safe choice: HTTPS only, hidden from scripts and not sent on cross-site POST requests.

Property	Type	Default	Description
CookieName	string	'sgcsid'	Name of the session cookie.
CookiePath	string	'/'	Path attribute of the cookies.
CookieDomain	string	''	Domain attribute. Empty writes a host-only cookie.
CookieSecure	Boolean	True	Secure attribute: the cookies only travel over HTTPS.
CookieHttpOnly	Boolean	True	HttpOnly attribute: page scripts cannot read the cookies.
CookieSameSite	TsgcHTMLCookieSameSite	hssLax	hssStrict, hssLax or hssNone (which always adds Secure).

Timeouts and Remember Me

What it does: *IdleTimeout* ends a session that was not used for a while, *AbsoluteTimeout* ends it a fixed time after the sign-in, and *RememberMeDays* lets a user who checked the remember box come back signed in after the session is gone.

Property	Type	Default	Description
IdleTimeout	Integer	1800	Seconds without requests before a session expires. 0 disables the check.
AbsoluteTimeout	Integer	43200	Seconds after the sign-in before a session expires, also the Max-Age of the session cookie. 0 disables the check and makes the cookie a browser-session cookie.
RememberMeDays	Integer	0	Lifetime of the remember-me token in days, up to 3650. 0 turns remember-me off.

Remember-CookieN-ame	string	'sgcrem'	Name of the remember-me cookie.

Endpoints

What it does: the paths the HTMX engine uses for its built-in login and logout endpoints and for its redirects.

Property	Type	De-fault	Description
Login-Path	string	'/login'	POST target of the built-in login and redirect target for anonymous re-quests to gated routes. The GET page is your own route.
Logout-Path	string	'/lo-gout'	POST target of the built-in logout.
After-Login-Path	string	'/'	Where a successful login goes when the form has no valid next field.

CSRF

What it does: every session has a random token. With *CSRFProtection* on, the engine refuses a request that has a session, uses a method other than GET, HEAD or OPTIONS and does not send the token back.

Property	Type	Default	Description
CSRFProtection	Boolean	True	Enables the check in the HTMX engine.
CSRFFieldName	string	'csrf_token'	Form field or query parameter that carries the token.
CSRFHeader-Name	string	'X-CSRF-Token'	Header that carries the token. sgcHTMX.min.js sends it for htmx requests.

Login Throttling

What it does: *LoginRateLimit* locks a client IP or a user name after too many failed logins in a row.

Property	Type	Default	Description
LoginRateLimit.MaxAttempts	Integer	5	Failures that lock the IP or the user name. 0 disables the limit.
LoginRateLimit.WindowSeconds	Integer	300	Failures only add up within this many seconds.
LoginRateLimit.LockoutSeconds	Integer	900	How long the lock lasts.

Events

Event	Type	Description
OnAuthenticate	TsgcHTMLAuthenticateEvent	Checks a user name and a password and returns the user id, display name and roles.
OnLoadUser	TsgcHTMLLoadUserEvent	Loads a user again when a remember-me cookie is redeemed.
OnSignIn	TsgcHTMLSessionEvent	Fired after a new session has been created.
OnSignOut	TsgcHTMLSessionEvent	Fired after a session has been ended by SignOut.

Methods

- **CurrentSession(aCookieHeader, aRemoteIP, aUserAgent, aSetCookies): TsgcHTMLSession**: the session of a request, or nil. Checks the timeouts and redeems remember-me cookies.
- **Authenticate(aUser, aPassword, aRemoteIP, var aUserID, var aDisplayName, aRoles): Boolean**: fires OnAuthenticate with login throttling.
- **SignIn(aUserID, aDisplayName, aRoles, aRemember, aOldSessionID, aRemoteIP, aUserAgent, aSetCookies): TsgcHTMLSession**: creates a new session with a new id and a new CSRF token.
- **SignOut(aSessionID, aSetCookies)**: ends a session and returns the delete cookies.
- **SignOutEverywhere(aUserID)**: ends every session of a user.
- **IssueCSRF(aSession): string** and **ValidateCSRF(aSession, aToken): Boolean**: read and check the CSRF token of a session.
- **IsLoginLocked(aRemoteIP, aUser): Boolean**: tells whether the login rate limit is locking the IP or the user name.
- **GetCookieValue(aCookieHeader, aName): string** and **BuildSetCookie(...): string**: class functions to read a cookie and to build a Set-Cookie header value.

Passwords

The unit also contains two functions to store passwords safely. **sgcHTMLPasswordHash**(aPassword, alterations = 210000) returns `pbkdf2-sha256$<iterations>$<salt>$<hash>`: PBKDF2 with SHA-256 over the UTF-8 bytes of the password, a random 16 byte salt and a 32 byte key, salt and key in base64. It raises an exception when alterations is below 1 or above 10000000. **sgcHTMLPasswordVerify**(aPassword, aHash) compares in constant time and returns False, without raising, for a malformed hash. A hash keeps verifying after you raise the iteration count for new passwords.

```
// when the user sets a password
UnicodeString vHash = sgcHTMLPasswordHash(vNewPassword);

// in OnAuthenticate
aAccept = sgcHTMLPasswordVerify(aPassword, vStoredHash);
```

Request Context

sgcHTMLSetRequestAuth(aAuth), **sgcHTMLClearRequestAuth** and **sgcHTMLRequestAuth** keep the TsgcHTMLAuth that serves the request of the calling thread, in a thread variable. The HTMX engine sets it, together with the request session, while a page or a route is rendered, so that components without an Auth of their own (TsgcHTMLComponent_Login, TsgcHTMLComponent_Site) find the login and logout paths and the CSRF field name. When you serve pages from your own handlers, set both yourself and clear them in a finally block. The pointer is not owned.

Thread Safety

Configure the properties before the server starts. After that every method can be called from any number of threads at the same time: the session store is thread-safe and the login attempt table has its own lock. The events fire in the thread that serves the request.

See Also

- Sessions, CSRF and Authentication
- Routing
- TsgcHTMLSessionStore_Memory
- TsgcHTMLSessionStore_File
- TsgcHTMX_Engine_Server
- TsgcHTMLComponent_Login

Reference

- Properties: 18 properties available on the component.
- Methods: 10 public methods.
- Events: 4 published events.

TsgcHTMLSessionStore_Memory

TsgcHTMLSessionStore_Memory keeps the sessions and remember-me tokens of TsgcHTMLAuth in the memory of the process, with an optional background thread that removes the expired ones.

Introduction

TsgcHTMLSessionStore_Memory is the session store for applications served by a single process. TsgcHTMLAuth keeps its sessions and its remember-me tokens here. Everything lives in the memory of the process, so after a restart every user is anonymous again and the remember-me cookies no longer work. When several processes serve the same site (an ISAPI module in a web garden, CGI executables, several servers behind a load balancer) use TsgcHTMLSessionStore_File, or a descendant of TsgcHTMLSessionStore that keeps the data in your database.

The store is thread-safe: every public method is serialized by an internal lock, so one store can be shared by any number of threads. The component lives in the unit **sgcHTML_Session** and is registered on the **SGC HTML** palette page.

Quick Start

1. Drop a **TsgcHTMLSessionStore_Memory** (or create it at runtime).

2. Give it the same **IdleTimeout** and **AbsoluteTimeout** as the TsgcHTMLAuth and set **SweepInterval** to remove expired sessions in the background.

3. Assign it to the **SessionStore** property of the TsgcHTMLAuth.

```
oStore = new TsgcHTMLSessionStore_Memory(this);
oStore->IdleTimeout = oAuth->IdleTimeout;
oStore->AbsoluteTimeout = oAuth->AbsoluteTimeout;
oStore->SweepInterval = 60; // sweep every minute
oAuth->SessionStore = oStore;
```

Background Sweeper

What it does: TsgcHTMLAuth refuses an expired session as soon as it reads it, but the session stays in memory until a sweep removes it. With *SweepInterval* greater than 0 a background thread calls **Sweep**(IdleTimeout, AbsoluteTimeout) at that interval. The thread never runs at design time and is stopped when the component is freed.

Property	Type	Default	Description
SweepInterval	Integer	0	Seconds between two sweeps. 0 disables the thread.
IdleTimeout	Integer	1800	Idle timeout used by the sweeper. Keep it equal to TsgcHTMLAuth.IdleTimeout.

AbsoluteTime-out	Inte-ger	43200	Absolute timeout used by the sweeper. Keep it equal to TsgcHTMLAuth.AbsoluteTimeout.

Methods

What it does: the methods are inherited from the abstract **TsgcHTMLSessionStore** and are the same for every store. TsgcHTMLAuth calls them for you; call them directly for administration tasks.

- **Get(aID): TsgcHTMLSession**: a copy of the session, owned by the caller, or nil.
- **Put(aSession)**: stores a copy of the session. The ID must not be empty.
- **Delete(aID): Boolean** and **Touch(aID): Boolean**: remove a session, or move its LastSeen to now.
- **Sweep(aIdleTimeout, aAbsoluteTimeout): Integer**: removes the expired sessions and remember-me tokens.
- **DeleteUser(aUserID): Integer**: removes every session and remember-me token of a user.
- **PutRememberToken**, **GetRememberToken** and **DeleteRememberToken**: low level storage of the remember-me tokens.

TsgcHTMLSession

What it is: the signed-in user session, a plain object declared in sgcHTML_Session. The stores never hand out the instance they keep: Get returns a copy and Put stores a copy, so the thread that owns a session can read and change it without any lock. Every date is UTC.

Property	Type	Description
ID	string	Session id, the value of the session cookie: 43 base64url characters assigned by SignIn.
UserID	string	Stable id of the signed-in user.
Display-Name	string	Name shown by the components (Site header, OAuthCallback).
Roles	TStringList	Roles of the user. HasRole compares them without case.
Values	TStringList	Free name=value pairs for your own data, for example `Values.Values['username']`.
CSRFToken	string	Token that state-changing requests must send back (see TsgcHTMLAuth.IssueCSRF).
CreatedAt	TDate-Time	Sign-in time, UTC. The absolute timeout counts from here.
LastSeen	TDate-Time	Time of the last request, UTC. The idle timeout counts from here.
RemoteIP	string	Client IP at sign-in.

UserAgen-tHash	string	Lowercase hex SHA-256 of the User-Agent at sign-in, empty when there was none.
Remem-berSelector	string	Selector of the remember-me token issued with the session, so signing out revokes it too. Empty when no token was issued.

- **Clone: TsgcHTMLSession**: returns a new copy, owned by the caller.
- **Assign(aSource)**: copies every member of another session.
- **HasRole(aRole): Boolean**: True when Roles contains aRole, compared without case. False for an empty role.
- **IsExpired(aNowUTC, aIdleTimeout, aAbsoluteTimeout): Boolean**: True when the session was idle longer than aIdleTimeout seconds or exists for longer than aAbsoluteTimeout seconds. 0 disables a check.

During a request served by the HTMX engine, the session of the request is available as **TsgcHTMXRequest.Session** and through **sgcHTMLRequestSession**. That object belongs to the engine and is freed when the request ends: do not keep it, and call `Auth.SessionStore.Put` to persist a change.

Custom Stores

To keep the sessions in a database, descend from **TsgcHTMLSessionStore** and override the protected methods DoGet, DoPut, DoDelete, DoTouch, DoSweep, DoDeleteUser, DoPutRememberToken, DoGetRememberToken and DoDeleteRememberToken. They are always called with the store lock held. DoGet must return a new object that the caller owns, DoPut must store a copy, and ids must be compared case-sensitively. The protected virtual function NowUTC returns the clock used by Touch and Sweep, override it to test expiry without waiting.

Helper Functions

- **sgcHTMLNewToken**(aByteCount = 32): base64url text, without padding, of aByteCount bytes from the random generator of the platform. A value below 1 means 32.
- **sgcHTMLNewSessionID**: a new session id, 32 random bytes in base64url (43 characters).
- **sgcHTMLSHA256Hex**(aValue): lowercase hex SHA-256 of the UTF-8 bytes of aValue.
- **sgcHTMLSetRequestSession**(aSession), **sgcHTMLClearRequestSession** and **sgcHTMLRequestSession**: the session of the request served by the calling thread, a thread variable. It is nil for anonymous requests and outside a request. The pointer is not owned: whoever sets it must clear it before the session is freed.

See Also

- TsgcHTMLAuth
- TsgcHTMLSessionStore_File
- Sessions, CSRF and Authentication

Reference

- Properties: 3 properties available on the component.
- Methods: 9 public methods.
- Events: this component has no published events.

TsgcHTMLSessionStore_File

TsgcHTMLSessionStore_File keeps each session and each remember-me token of TsgcHTMLAuth in its own JSON file, so several processes can share the signed-in users.

Introduction

TsgcHTMLSessionStore_File keeps every session and every remember-me token of TsgcHTMLAuth in its own JSON file inside **Folder**. Use it when several processes serve the same site, for example an ISAPI module in a web garden, CGI executables or several servers that share a folder, or when the signed-in users must survive a restart of the application.

Sessions are stored as `s_<hash>.json` and remember-me tokens as `r_<hash>.json`, where the hash is the lowercase hex SHA-256 of the session id or of the token selector. A directory listing does not reveal any session id, and an id read from a cookie can never point outside the folder. The session id itself is not written into the file.

Every write goes to a temporary file that then replaces the target in one rename (MoveFileEx on Windows, rename on the other systems), so a reader never sees a half written file; on Windows a reader retries for a short time while another process replaces the file. Inside one process every call is serialized by a lock. Between processes, when two of them update the same session at the same time, the last write wins. The component lives in the unit **sgcHTML_Session** and is registered on the **SGC HTML** palette page.

Quick Start

1. Drop a **TsgcHTMLSessionStore_File** (or create it at runtime) and set **Folder** to the same folder in every process.

2. Assign it to the **SessionStore** property of the TsgcHTMLAuth.

3. Call **Sweep** from time to time: this store has no background thread.

```
oStore = new TsgcHTMLSessionStore_File(this);
oStore->Folder = "C:\\ProgramData\\MyApp\\sessions";
oAuth->SessionStore = oStore;

// from a timer, every few minutes
oStore->Sweep(oAuth->IdleTimeout, oAuth->AbsoluteTimeout);
```

Storage

What it does: *Folder* holds the files. It is created, with its parents, the first time a file is written. Every method raises the exception `TsgcHTMLSessionStore_File: Folder is empty` while it is empty, and a file that cannot be written raises `TsgcHTMLSessionStore_File: cannot write <file>`. *TouchInterval* limits how often a request rewrites its session file.

Property	Type	De-fault	Description

Folder	string	''	Folder of the session and remember-me files. Only the account that runs the host may read it.
TouchInterval	Integer	60	Touch only rewrites a session file when its stored LastSeen is at least this many seconds old. 0 writes on every Touch. It is also the precision of the idle timeout.

File Format

The files are UTF-8 JSON. A session file has the members `userId`, `displayName`, `roles` (array), `values` (array of name=value), `csrfToken`, `createdAt`, `lastSeen`, `remoteIp`, `userAgentHash` and `rememberSelector`. A token file has `userId`, `verifierHash` and `expiresAt`. Dates are written as `yyyy-mm-ddThh:nn:ss.zzzZ` in UTC. A missing, unreadable or damaged file is treated as an unknown id; a session file counts as damaged when it has no userId, createdAt or lastSeen, or when one of those dates cannot be read. For Sweep, a token file is damaged when it has no userId, verifierHash or valid expiresAt.

Performance and Security

- Every request that has a session reads its file, and rewrites it at most once every *TouchInterval* seconds (60 by default). DeleteUser and Sweep read every file of the folder. Keep the folder on a fast disk, and prefer TsgcHTMLSessionStore_Memory when a single process serves the site.
- Nothing removes old files automatically. Call **Sweep** regularly with the timeouts of the TsgcHTMLAuth. Besides the expired sessions and tokens, it removes damaged files once their last write is older than the absolute timeout (the idle timeout when that is 0, one day when both are 0), and .tmp files older than one hour, left by writes that never completed.
- The files contain user ids, display names, roles, CSRF tokens and remember-me verifier hashes. Restrict the folder to the account that runs the host and never place it under the web root.

Methods

What it does: the methods are inherited from **TsgcHTMLSessionStore** and behave as described in TsgcHTMLSessionStore_Memory: Get, Put, Delete, Touch, Sweep, DeleteUser, PutRememberToken, GetRememberToken and DeleteRememberToken. The sessions are TsgcHTMLSession objects.

See Also

- TsgcHTMLAuth
- TsgcHTMLSessionStore_Memory
- Sessions, CSRF and Authentication

Reference

- Properties: 2 properties available on the component.
- Methods: 9 public methods.
- Events: this component has no published events.

TsgcHTMLComponent_Login

TsgcHTMLComponent_Login renders a Bootstrap username / password sign-in form, optionally wrapped as a card, a centered card or a full-page layout, and can emit a complete HTML document ready to serve.

Introduction

TsgcHTMLComponent_Login builds the markup for a classic credentials login screen: a title, an optional subtitle and logo, a username field, a password field, an optional "remember me" checkbox, a submit button and an optional footer link. You set the texts and the form target through properties and read the generated HTML through the inherited **HTML** property, or call **GetFullPageHTML** to obtain a standalone HTML page that already links Bootstrap.

The form posts to the URL in **FormAction** using the verb in **FormMethod**. Field names are fixed (`username`, `password` and `remember`) so your server-side handler always knows what to read.

Quick Start

1. Drop a **TsgcHTMLComponent_Login** component on the form (or create it at runtime).

2. Set the **FormAction**, the **Title** and (optionally) the **LoginStyle**.

3. Read the **HTML** property and serve it, or call **GetFullPageHTML** for a complete page.

```
TsgcHTMLComponent_Login *oLogin = new TsgcHTMLComponent_Login(NULL);
try
{
  oLogin->FormAction = "/login";
  oLogin->Title = "Sign in";
  oLogin->LoginStyle = lsFullPage;
  oLogin->SetLogoText("Acme");
  html = oLogin->GetFullPageHTML("Sign in to Acme");
}
__finally
{
  delete oLogin;
}
```

Form Target

What it does: *FormAction* and *FormMethod* control where the credentials are posted. *FormID* gives the generated `<form>` a stable id. *ShowRememberMe* adds a "remember me" checkbox whose value is posted as `remember`. *Auth* links the form to a TsgcHTMLAuth: with a FormAction that is empty or left at its default `/login` the form posts to Auth.LoginPath, and when the request has a session the form carries its CSRF token in a hidden field. *NextURL* adds a hidden `next` field that tells the built-in login endpoint where to go after the sign-in (see Sessions, CSRF and Authentication).

Property	Type	Default	Description

FormAction	string	'/login'	URL the credentials are posted to. With an Auth, an empty value or /login posts to Auth.LoginPath.
FormMethod	string	'POST'	HTTP verb of the form (usually POST).
FormID	string	'sgcLoginForm'	The id attribute of the generated <form> element.
ShowRememberMe	Boolean	False	Adds a "remember me" checkbox posted as `remember`.
Auth	TsgcHTMLAuth	nil	Authentication the form posts to (Auth.LoginPath, when FormAction is empty or /login) and takes its CSRF token from. The request auth is used when it is nil.
NextURL	string	''	Relative path sent in the hidden next field. Other values render no field.

Texts and Labels

What it does: *Title* and *Subtitle* head the form. *UserLabel* / *UserPlaceholder* and *PasswordLabel* / *PasswordPlaceholder* label and hint the two input fields. *ButtonText* sets the submit caption. *ErrorMessage* and *SuccessMessage* render a Bootstrap alert above the form when not empty.

Property	Type	Default	Description
Title	string	(locale)	Heading drawn above the form.
Subtitle	string	''	Optional sub-heading below the title.
UserLabel	string	(locale)	Label for the username field.
UserPlaceholder	string	(locale)	Placeholder text for the username field.
PasswordLabel	string	(locale)	Label for the password field.
PasswordPlaceholder	string	(locale)	Placeholder text for the password field.
ButtonText	string	(locale)	Caption of the submit button.
ErrorMessage	string	''	When set, shows a danger alert above the form.
SuccessMessage	string	''	When set, shows a success alert above the form.

Field Values and Autocomplete

What it does: *UserValue* and *PasswordValue* prefill the two inputs, for a demo or kiosk account. *UserAutocomplete* and *PasswordAutocomplete* tell the password manager of the browser what the fields are; an empty value leaves the attribute out.

Property	Type	Default	Description
UserValue	string	''	Value of the username input.
PasswordValue	string	''	Value of the password input. It is visible in the page source, so never use it for a real password.
UserAutocomplete	string	'username'	autocomplete attribute of the username input.
PasswordAutocomplete	string	'current-password'	autocomplete attribute of the password input.

Appearance and Layout

What it does: *LoginStyle* wraps the form as a plain card, a vertically centered card or a full-page layout. *ButtonStyleEnum*, *ButtonClass*, *CSSClass*, *MaxWidth*, *MinHeight* and *BackgroundClass* tune the look. *LogoHTML* / *LogoText* and *FooterHTML* / *FooterLinkText* / *FooterLinkURL* add a logo and a footer; the **SetLogoText** and **SetFooterText** helpers wrap plain text in the expected markup. *TitleTag* and *TitleClass* choose the heading element of the title, *FooterClass* the element that wraps FooterHTML, and *ButtonBlock* whether the button sits in a d-grid block.

Property	Type	Default	Description
LoginStyle	TsgcHTMLLoginStyle	lsCentered	Layout wrapper: lsCard, lsCentered or lsFullPage.
ButtonStyleEnum	TsgcHTMLButtonStyle	bsPrimary	Bootstrap style of the submit button.
ButtonClass	string	''	Explicit CSS class for the button (overrides ButtonStyleEnum).
CSSClass	string	''	Extra CSS class added to the wrapping card.
MaxWidth	string	'400px'	Maximum width of the card.
MinHeight	string	'80vh'	Minimum height used by the lsCentered layout.
BackgroundClass	string	'bg-light'	Background CSS class used by the lsFullPage layout.
LogoHTML	string	''	Raw HTML drawn as the logo above the title.

LogoText	string	''	Plain logo text (used when LogoHTML is empty).
FooterHTML	string	''	Raw HTML drawn below the form.
FooterLinkText	string	''	Footer link caption (used when FooterHTML is empty).
FooterLinkURL	string	''	Footer link URL paired with FooterLinkText.
TitleTag	string	'h3'	Element of the title heading. Empty renders no title.
TitleClass	string	'text-center mb-3'	class attribute of the title heading.
FooterClass	string	'text-center mt-3'	class of the div around FooterHTML. Empty writes FooterHTML on its own.
ButtonBlock	Boolean	True	Wraps the submit button in a d-grid block.

Full Page Output

What it does: GetFullPageHTML returns a complete HTML document that links the Bootstrap stylesheet from *BootstrapCSSPath* and the Bootstrap script from *BootstrapJSPath*, with the login form in the body. Use the inherited **HTML** property instead when you only need the form fragment to embed in your own page.

Property	Type	Default	Description
BootstrapCSSPath	string	'/bootstrap.min.css'	Stylesheet path linked by GetFullPageHTML.
BootstrapJSPath	string	'/bootstrap.bundle.min.js'	Script path linked by GetFullPageHTML.
HtmlLang	string	'en'	lang attribute on the <html> element produced by GetFullPageHTML.

Reference

- Properties — 38 properties available on the component.
- Methods — 3 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_SocialLogin

TsgcHTMLComponent_SocialLogin renders a column or row of "Continue with" buttons for OAuth providers (Google, Facebook, Apple, GitHub, Microsoft, Twitter, LinkedIn or a custom one), each linking to the provider's authorization URL.

Introduction

TsgcHTMLComponent_SocialLogin builds the markup for social / OAuth sign-in. You add one entry per provider to the **Providers** collection (or call **AddProvider**), supplying at least the client id and the redirect URI. For each entry the component knows the provider's brand color, icon, default authorization endpoint and default scope, so it can emit a styled button whose link is the fully formed OAuth authorization URL.

Each provider entry is a **TsgcHTMLSocialProviderItem** exposing *Provider*, *ClientID*, *RedirectURI*, *Scope*, *AuthURL*, *ButtonText*, *CustomColor*, *CustomIcon*, *IconURL* and *State*. Read the generated markup through the inherited **HTML** property.

Quick Start

1. Drop a **TsgcHTMLComponent_SocialLogin** component on the form (or create it at runtime).

2. Add one or more providers with **AddProvider**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_SocialLogin *oSocial = new TsgcHTMLComponent_SocialLogin(NULL);
try
{
  oSocial->Title = "Sign in";
  oSocial->AddProvider(spGoogle, "my-google-client-id", "https://app.example.com/oauth/google");
  oSocial->AddProvider(spGitHub, "my-github-client-id", "https://app.example.com/oauth/github");
  html = oSocial->HTML;
}
__finally
{
  delete oSocial;
}
```

Providers

What it does: The **Providers** collection holds one **TsgcHTMLSocialProviderItem** per button. Add items at design time, assign them directly, or call **AddProvider** at runtime. For each item the component derives the authorization URL from *AuthURL* (or the provider default), the requested *Scope* (or the provider default), the brand color and the icon, then renders an anchor styled as a button.

Property	Type	Default	Description
Providers	TsgcHTMLSocial-ProviderItems	(empty)	Collection of provider buttons rendered in order.

Heading and Divider

What it does: *Title* and *Subtitle* head the button block. *ShowDivider* and *DividerText* draw an "or continue with" separator below the buttons. *LoginID* gives the wrapping `<div>` a stable id.

Property	Type	Default	Description
Title	string	(locale)	Heading drawn above the buttons.
Subtitle	string	'Choose your login method'	Muted sub-heading below the title.
ShowDivider	Boolean	True	Shows the "or continue with" divider below the buttons.
DividerText	string	'or continue with'	Text drawn in the divider.
LoginID	string	'sgcSocialLogin'	The id of the wrapping element.

Layout and Buttons

What it does: *Layout* stacks the buttons vertically or lays them out in a wrapping row. *ShowIcons* toggles the provider icons. *MaxWidth*, *ButtonPadding* and *ButtonBorderRadius* tune the button block size and shape.

Property	Type	Default	Description
Layout	TsgcHTMLSocialLoginLayout	slVertical	Button arrangement: slVertical or slHorizontal.
ShowIcons	Boolean	True	Shows or hides the provider icons on the buttons.
MaxWidth	string	'400px'	Maximum width of the centered button block.
ButtonPadding	string	'12px 20px'	CSS padding applied to each button.
ButtonBorderRadius	string	'8px'	CSS border radius applied to each button.

Output

The inherited **HTML** property returns a `<div>` with the optional title and subtitle, the list of provider buttons and the optional divider. Each button is an anchor whose `href` is the provider authorization URL, so clicking it starts the OAuth flow. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 11 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_OAuthCallback

TsgcHTMLComponent_OAuthCallback renders the landing page shown after an OAuth provider redirects back to your app: a loading spinner, a success card with the signed-in user and an optional auto-redirect, or an error card.

Introduction

TsgcHTMLComponent_OAuthCallback builds the result screen for the OAuth redirect endpoint. The **Status** property selects one of three states: a loading state while you exchange the authorization code, a success state that greets the user and (optionally) redirects to a destination after a delay, and an error state that shows a message and a "try again" link. The matching markup is returned through the inherited **HTML** property.

For the common cases you do not need to create an instance at all: the static **BuildSuccess**, **BuildError** and **BuildLoading** class functions create a configured component internally and return the HTML in one call.

Quick Start

1. In your OAuth redirect handler, exchange the authorization code for the user's profile.

2. Call **BuildSuccess** (or **BuildError** / **BuildLoading**) with the provider name and details.

3. Serve the returned HTML to the browser.

```
// After a successful token exchange
html = TsgcHTMLComponent_OAuthCallback::BuildSuccess("Google", "Jane Doe",
  "/dashboard", "jane@example.com");
```

Status and Identity

What it does: *Status* selects the loading, success or error card. *ProviderName* names the provider in every state. In the success state, *UserName*, *UserEmail* and *UserAvatar* identify the signed-in user (shown only when *ShowUserInfo* is True), and *ErrorMessage* carries the detail shown in the error state. When *UserName* is empty, the success state shows the DisplayName of the request session.

Property	Type	Default	Description
Status	TsgcHTMLOAuthCallbackStatus	csLoading	Which card to render: csSuccess, csError or csLoading.
ProviderName	string	''	Name of the OAuth provider shown in the message.
UserName	string	''	Display name of the signed-in user. When empty, the DisplayName of the request session is used.

UserEmail	string	''	Email of the signed-in user.
UserAvatar	string	''	Avatar image URL; a letter badge is drawn when empty.
ErrorMessage	string	''	Detail message shown in the error card.
ShowUserInfo	Boolean	True	Shows the user name, email and avatar on success.
AvatarSize	Integer	48	Avatar size in pixels.

Redirect

What it does: On success, *RedirectMethod* decides whether to auto-redirect, show only a continue button, or do nothing. *RedirectURL* is the destination and *RedirectDelay* is the delay in seconds before an automatic redirect.

Property	Type	Default	Description
RedirectMethod	TsgcHTMLOAuthRedirectMethod	rmAutoRedirect	How to leave the page: rmAutoRedirect, rmButtonOnly or rmNone.
RedirectURL	string	'/dashboard'	Destination after a successful sign-in.
RedirectDelay	Integer	3	Seconds to wait before an automatic redirect.

Icons and Appearance

What it does: The success and error cards each draw a large icon. *SuccessIconHTML* / *SuccessIconText* and *ErrorIconHTML* / *ErrorIconText* set the glyph; *SuccessIconColor* / *SuccessIconColorEnum* and *ErrorIconColor* / *ErrorIconColorEnum* set the color (an explicit CSS color wins over the enum). *CallbackID* and *MaxWidth* identify and size the card.

Property	Type	Default	Description
SuccessIconHTML	string	''	Raw markup for the success icon (overrides SuccessIconText).
SuccessIconText	string	'✓'	Glyph for the success icon when SuccessIconHTML is empty.
SuccessIconColor	string	''	Explicit CSS color for the success icon.

SuccessIconColorEnum	TsgcHTMLColor	hcSuccess	Themed color for the success icon when no CSS color is set.
ErrorIconHTML	string	''	Raw markup for the error icon (overrides ErrorIconText).
ErrorIconText	string	'✗'	Glyph for the error icon when ErrorIconHTML is empty.
ErrorIconColor	string	''	Explicit CSS color for the error icon.
ErrorIconColorEnum	TsgcHTMLColor	hcDanger	Themed color for the error icon when no CSS color is set.
CallbackID	string	'sgcOAuthCallback'	The id of the card element.
MaxWidth	string	'450px'	Maximum width of the card.

Output

The inherited **HTML** property returns a Bootstrap card for the current **Status**. The success card may embed a small `<script>` that performs the timed redirect. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 21 properties available on the component.
- Methods — 3 public methods.
- Events — this component has no published events.

TsgcHTMLComponent_WebAuthnLogin

TsgcHTMLComponent_WebAuthnLogin renders passwordless passkey (WebAuthn) sign-in: register and authenticate buttons plus the browser-side script that drives the SimpleWebAuthn ceremonies against your endpoints.

Introduction

TsgcHTMLComponent_WebAuthnLogin builds a passkey panel: a title, a short description, an optional passkey icon, a "register passkey" button and a "sign in with passkey" button, plus an inline `<script>` that runs the WebAuthn registration and authentication ceremonies in the browser. The script posts to the **RegisterURL** and **AuthenticateURL** endpoints (each with an `/options` and a `/verify` step) and, on success, navigates to **CallbackURL**.

The generated script uses the `SimpleWebAuthnBrowser` helper (`startRegistration` / `startAuthentication`), which your page must load. Read the generated markup through the inherited **HTML** property, or use the static **Build** helper for the common case.

Quick Start

1. Drop a **TsgcHTMLComponent_WebAuthnLogin** component on the form (or create it at runtime).

2. Set the **Mode** and the register / authenticate endpoint URLs.

3. Read the **HTML** property and serve it (or call the static **Build** helper).

```
TsgcHTMLComponent_WebAuthnLogin *oWebAuthn = new TsgcHTMLComponent_WebAuthnLogin(NULL);
try
{
  oWebAuthn->Mode = wamBoth;
  oWebAuthn->RegisterURL = "/webauthn/register";
  oWebAuthn->AuthenticateURL = "/webauthn/authenticate";
  oWebAuthn->CallbackURL = "/dashboard";
  html = oWebAuthn->HTML;
}
__finally
{
  delete oWebAuthn;
}
```

Mode and Endpoints

What it does: *Mode* selects which buttons appear (register, authenticate or both). *RegisterURL* and *AuthenticateURL* are the base endpoints the script calls (each gets an `/options` and a `/verify` suffix). *CallbackURL* is where the browser goes after a successful authentication. *UsernameSelector* lets the script read a username from another input before registering.

Property	Type	Default	Description
Mode	TsgcHTMLWebAuthnMode	wamBoth	Which buttons to show: wamRegister, wamAuthenticate or wamBoth.

RegisterURL	string	'/webauthn/register'	Base endpoint for the registration ceremony.
AuthenticateURL	string	'/webauthn/authenticate'	Base endpoint for the authentication ceremony.
CallbackURL	string	'/dashboard'	Destination after a successful authentication.
UsernameSelector	string	''	CSS selector of an input whose value is sent when registering.

Texts and Buttons

What it does: *Title* and *Description* head the panel. *RegisterButtonText* / *AuthenticateButtonText* caption the two buttons and *RegisterButtonStyle* / *AuthenticateButtonStyle* set their Bootstrap variants. *ShowPasskeyIcon* and *PasskeyIconSize* control the key icon above the panel.

Property	Type	Default	Description
Title	string	(locale)	Heading of the passkey panel.
Description	string	(locale)	Muted description under the title.
RegisterButtonText	string	(locale)	Caption of the register button.
AuthenticateButtonText	string	(locale)	Caption of the authenticate button.
RegisterButtonStyle	TsgcHTMLButtonStyle	bsOutlinePrimary	Bootstrap style of the register button.
AuthenticateButtonStyle	TsgcHTMLButtonStyle	bsPrimary	Bootstrap style of the authenticate button.
ShowPasskeyIcon	Boolean	True	Shows the key icon above the panel.
PasskeyIconSize	string	'2.5rem'	CSS font size of the key icon.

Status, Script and Identity

What it does: *WebAuthnID* identifies the panel and its status element. *SuccessAlertClass* / *SuccessAlertStyle* and *ErrorAlertClass* / *ErrorAlertStyle* style the status messages the script writes (an explicit class wins over the style enum). *CustomScript* replaces the entire generated script with your own.

Property	Type	Default	Description

WebAuthnID	string	'sgcWebAuthn'	The id of the panel and the base of the status element id.
SuccessAlertClass	string	''	Explicit CSS class for the success status (overrides SuccessAlertStyle).
SuccessAlertStyle	TsgcHTMLAlertStyle	asSuccess	Alert style for the success status message.
ErrorAlertClass	string	''	Explicit CSS class for the error status (overrides ErrorAlertStyle).
ErrorAlertStyle	TsgcHTMLAlertStyle	asDanger	Alert style for the error status message.
CustomScript	string	''	Replaces the generated WebAuthn script entirely.

Output

The inherited **HTML** property returns a `<div>` with the icon, title, description, a status element, the enabled buttons and the WebAuthn `<script>`. The page must also load the `SimpleWebAuthnBrowser` helper for the ceremonies to run. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 19 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLComponent_UserManagement

TsgcHTMLComponent_UserManagement renders a user administration table with an avatar plus name and email, role badges, a status badge, the last login and a per-row Actions drop-down menu.

Introduction

TsgcHTMLComponent_UserManagement emits a responsive HTML table where every row is one user account. The first column shows an avatar (an image when **AvatarURL** is set, otherwise the initial letter of the name) together with the display name and the email address. The remaining columns show the role badges, a status badge, the last login and an Actions drop-down menu.

Choosing an item in the Actions menu emits a flat JSON payload over the WebSocket channel through `window.sgcHTMX.send`, and also dispatches a DOM CustomEvent named `sgcUserManagement:action`. The host application receives that message, authorizes it, and then calls **ProcessUserAction**, which raises **OnUserAction** on the server.

The user list can be filled by hand through the **Users** collection or **AddUser**, loaded from any TDataSet with **LoadFromDataSet**, or bound to a **DataSource** and refreshed automatically using the column mapping given by the **Field* properties.

Quick Start

1. Drop a **TsgcHTMLComponent_UserManagement** component on the form (or create it at runtime) and add the users with **AddUser**, **LoadFromDataSet** or a bound **DataSource**.

2. Handle **OnUserAction**. When an action message arrives over the WebSocket channel, authorize it on the server first, and only then call **ProcessUserAction**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_UserManagement *oUsers = new TsgcHTMLComponent_UserManagement(NULL);
try
{
  oUsers->AllowImpersonate = false;
  oUsers->AllowDelete = false;

  TsgcHTMLUser *oUser = oUsers->AddUser("1", "jsmith", "jsmith@example.com");
  oUser->DisplayName = "John Smith";
  oUser->Roles = "admin,editor";
  oUser->Status = usActive;
  oUser->LastLogin = "2026-07-09 08:12";

  html = oUsers->HTML;
}
__finally
{
  delete oUsers;
}
```

Security

These components render the user interface only. They do NOT enforce authorization.

Actions and grants arrive from the browser as client-supplied data, so a hostile client can forge any action, user id, role id or permission id, including ones that were never rendered, and including when **ReadOnly** is True (ReadOnly is a rendering hint, not a control).

The host application must verify on the server, for every message it receives, that the authenticated acting user is permitted to perform that action on that target, and it should audit impersonation, deletion and permission grants.

Hiding a button in the UI hides nothing from an attacker.

Applied to this component: setting **AllowImpersonate** or **AllowDelete** to False only removes the menu item from the rendered HTML. It does not stop a client from sending the `userImpersonate` or `userDelete` action over the WebSocket channel. Gate both on the server, and log them.

Role check. Set *RequiredRole* to render the table, and to let **ProcessUserAction** do anything, only for a request session that has that role (the session set by the HTMX engine when it has an Auth). Any other request gets a danger alert with *AccessDeniedText*. The check covers the role only: every message still needs the server side authorization described above. See Sessions, CSRF and Authentication.

Property	Type	Default	Description
RequiredRole	string	''	Role the request session must have. Empty disables the check.
AccessDeniedText	string	'Access denied.'	Text of the alert shown instead of the table.

Columns and Layout

What it does: The User column is always rendered. *ShowRoles*, *ShowStatus*, *ShowLastLogin* and *ShowActions* add or remove the other four columns. *Striped*, *Bordered* and *Hover* select the table styling, *EmptyText* is the message shown when there is no user to list, and *TableID* is the id of the generated table element.

Property	Type	Default	Description
ShowRoles	Boolean	True	Shows the Roles column, one badge per role name.
ShowStatus	Boolean	True	Shows the Status column as a coloured badge.
ShowLastLogin	Boolean	True	Shows the Last login column.
ShowActions	Boolean	True	Shows the Actions column with the per-row drop-down menu.
Striped	Boolean	True	Alternating row background.
Bordered	Boolean	False	Draws borders around every cell.
Hover	Boolean	True	Highlights the row under the mouse pointer.

EmptyText	string	'No users found.'	Text of the single centred row rendered when Users is empty.
TableID	string	''	The id of the generated table element. Empty means a unique id is generated on the first render.

TableID is not hardcoded to a literal any more. Leave it empty and a unique element id is generated automatically the first time the component renders, and reused for the life of the instance, so two instances on the same page never collide and pushed fragments keep targeting the element this instance actually rendered. Set it explicitly if you need a stable, known id (for example to reach the element from your own JavaScript, or to recognise the table in the action payload). The generated id is `sgcUserManagement` plus a process-unique number, such as `sgcUserManagement2`. The search box and the pager derive their ids from it, as `<TableID>_search` and `<TableID>_pager`.

The Status column renders **TsgcHTMLUserStatus** as a badge: *usActive* is green, *usInvited* is blue (info), *usSuspended* is amber (warning) and *usDisabled* is grey (secondary).

Actions Menu

What it does: Every row ends with an Actions drop-down menu. Edit, Disable or Enable, and Reset password are always present. The Impersonate item is rendered only when *AllowImpersonate* is True, and the Delete item only when *AllowDelete* is True (Delete is preceded by a divider and styled in red). The Disable and Enable item flips with the user's *Enabled* flag: an enabled user gets a Disable item, a disabled user gets an Enable item.

Choosing an item sends the flat JSON payload `{action, table, userId}` through `window.sgcHTMX.send` and dispatches the DOM CustomEvent `sgcUserManagement:action` with the same payload in its detail. The Add button in the toolbar is a `data-sgc-ws-send` form and sends `{action:"userAdd", table}`, without a userId.

These are the action strings emitted by the component. The list is the contract between the browser and the server.

Action	Emitted by
userAdd	The Add button in the toolbar, when ShowAddButton is True.
userEdit	The Edit menu item.
userDisable	The Disable menu item, rendered when the user's Enabled is True.
userEnable	The Enable menu item, rendered when the user's Enabled is False.
userResetPassword	The Reset password menu item.
userImpersonate	The Impersonate menu item, rendered only when AllowImpersonate is True.
userDelete	The Delete menu item, rendered only when AllowDelete is True.

On the server side the round trip is: your WebSocket message handler receives the payload, checks that the authenticated acting user is allowed to perform that action on that target user, and then calls **ProcessUserAction**, which raises **OnUserAction**. The only check the component makes is *RequiredRole*, against the role of the request session.

```
// 1. the message handler authorizes first
if (AdminCanManage(vActingUserId, vAction, vUserId))
```

```
    oUsers->ProcessUserAction(vAction, vUserId);

  // 2. OnUserAction then does the work
  void __fastcall TForm1::oUsersUserAction(TObject *Sender,
    const UnicodeString aAction, const UnicodeString aUserId)
  {
    if (aAction == "userDelete")
    {
      AuditLog("delete", aUserId);
      DeleteUser(aUserId);
    }
  }
```

Search and Paging

What it does: *ShowSearch* renders a search box above the table that filters the rows in the browser, with no round trip to the server, and *SearchPlaceholder* is its placeholder text. *ShowAddButton* renders the Add button on the right of the same toolbar, captioned with *AddCaption*. *PageSize* greater than zero turns on client-side pagination and adds a pager under the table.

Property	Type	Default	Description
ShowSearch	Boolean	True	Renders the client-side search box.
SearchPlace-holder	string	'Search users...'	Placeholder text of the search box.
ShowAddButton	Boolean	True	Renders the Add button, which sends the userAdd action.
AddCaption	string	'Add user'	Caption of the Add button.
PageSize	Integer	0	Rows per page. Zero means no paging, all rows are shown.

The search filter matches the visible text of the row, so it covers the name, the email, the role names, the status caption and the last login. Paging is applied to the rows that survive the filter.

Data Binding

What it does: *Users* is the collection of **TsgcHTMLUser** items rendered by the table. Add items with **AddUser**, remove them all with **Clear**, or fill the collection from a dataset with **LoadFromDataSet**. Assigning a *DataSource* makes the component load the rows itself, using the **Field* properties as the column mapping, and reload them when *DataAutoRefresh* is enabled.

Each **TsgcHTMLUser** exposes *Id*, *UserName*, *DisplayName*, *Email*, *AvatarURL*, *Roles* (a comma-separated list of role names, for example `admin,editor`), *Status*, *LastLogin* and *Enabled*.

Property	Type	Default	Description
IdField	string	'id'	Dataset field mapped to TsgcHTMLUser.Id.

UserName-Field	string	'user-name'	Dataset field mapped to TsgcHTMLUser.UserName.
DisplayName-Field	string	''	Dataset field mapped to DisplayName. Empty means the column is not read and the user name is shown instead.
EmailField	string	'email'	Dataset field mapped to TsgcHTMLUser.Email.
AvatarURL-Field	string	''	Dataset field mapped to AvatarURL. Empty means the column is not read and the initials avatar is drawn.
RolesField	string	''	Dataset field mapped to Roles, a comma-separated list. Empty means the column is not read.
StatusField	string	''	Dataset field mapped to Status. Empty means the column is not read and every user is Active.
LastLogin-Field	string	''	Dataset field mapped to LastLogin. Empty means the column is not read.
EnabledField	string	''	Boolean dataset field mapped to Enabled. Empty means the column is not read and every user is enabled.

```
oUsers->StatusField = "state";
oUsers->LastLoginField = "last_login";
oUsers->LoadFromDataSet(qryUsers, "id", "username", "email", "roles", "state");
```

Rendering is XSS-safe. Names, emails and role names are HTML-encoded, the avatar URL is sanitized (`javascript:` and `data:text/html` are rejected) and attribute-encoded, and user ids only ever appear attribute-encoded inside `data-sgc-user`, never inside a JavaScript string literal. This is output encoding, it is not authorization. Keep the two points separate: safe encoding stops a stored script from running in another administrator's browser, it does nothing to stop a forged action arriving from a hostile client.

Output

The inherited **HTML** property returns the toolbar (the search box and the Add button), the responsive table, an optional pager `<div>` when **PageSize** is greater than zero, and an inline script that wires the Actions menu, the client-side search filter and the pager. Serve this string directly, or assign it to the BodyContent of a **TsgcHTMLTemplate_Bootstrap** to produce a full HTML document.

Reference

- Properties — 28 properties available on the component.
- Methods — 4 public methods.
- Events — 1 published event, raised when the host calls ProcessUserAction.

TsgcHTMLComponent_ImpersonateBanner

Renders the "you are impersonating X" warning bar, with a Stop button, that is shown while an administrator is acting as another user.

Introduction

TsgcHTMLComponent_ImpersonateBanner is a server-rendered warning bar. While an administrator is impersonating another user, the bar sits across the top of the page, states whose account is being used, and offers a single button that ends the impersonation.

The bar is a plain block of HTML. It needs no JavaScript of its own, and it uses only Bootstrap classes plus a small inline style when it is pinned to the top of the viewport. The Stop button works in one of two ways. When *StopURL* is set, the button is a link to that URL, which suits a classic request/response application with a dedicated stop endpoint. When *StopURL* is empty, the button submits a small **data-sgc-ws-send** form that sends the **userStopImpersonate** action over the WebSocket channel, which suits a live application driven by the sgcHTML WebSocket/HTMX engine.

Quick Start

1. Drop a **TsgcHTMLComponent_ImpersonateBanner** on the form, or create it in code.

2. Set **UserName** to the name of the user being impersonated, and set **StopURL** if you want the button to navigate to a stop endpoint instead of sending a WebSocket message.

3. Read the inherited **HTML** property and place the markup at the very start of the page body. For a one-line version, call the class function **Build**, which needs no instance.

```
TsgcHTMLComponent_ImpersonateBanner *oBanner = new TsgcHTMLComponent_ImpersonateBanner(NULL);
try {
  oBanner->UserName = "alice@example.com";
  oBanner->Color = hcWarning;
  oBanner->Sticky = true;
  UnicodeString vHTML = oBanner->HTML;
} __finally {
  delete oBanner;
}

// fast path: one line, no instance
UnicodeString vHTML = TsgcHTMLComponent_ImpersonateBanner::Build("alice@example.com");

// fast path with a stop endpoint
UnicodeString vHTML2 = TsgcHTMLComponent_ImpersonateBanner::Build("alice@example.com", "/admin/stop-impersona
```

Security

These components render the user interface only. They do NOT enforce authorization.

Actions and grants arrive from the browser as client-supplied data, so a hostile client can forge any action, user id, role id or permission id, including ones that were never rendered, and including when **ReadOnly** is True (ReadOnly is a rendering hint, not a control).

The host application must verify on the server, for every message it receives, that the authenticated acting user is permitted to perform that action on that target, and it should audit impersonation, deletion and permission grants.

Hiding a button in the UI hides nothing from an attacker.

For this component in particular, the banner is a visual reminder. It is not the impersonation mechanism and it is not a session control. The real impersonation state lives in the server-side session. A client can send **userStopImpersonate** at any time, and can equally just never render the banner, so never rely on the banner being present to decide whether impersonation is active. Ending an impersonation must be done on the server, and impersonation start and stop must both be audited.

Message and Caption

What it does: the text of the bar comes from *Message* and *UserName*, and the button caption comes from *StopCaption*. *Message* is plain text containing a single **%s** placeholder, which is replaced by the value of *UserName*. The substitution is textual, it is not a call to Format, so a user name that contains a **%** character, or a message that contains several placeholders, cannot raise a format exception. The whole resulting string is HTML encoded on output, so a hostile user name cannot inject markup.

Property	Type	Default	Description
UserName	string	”	Name of the user being impersonated. Replaces the %s placeholder in Message.
Message	string	You are currently impersonating %s.	Plain text of the bar. A single %s placeholder is replaced by UserName. An empty Message falls back to %s, so the bar shows the user name alone.
StopCaption	string	Stop impersonating	Caption of the button that ends the impersonation.

Stop Behaviour

What it does: *StopURL* decides how the Stop button behaves. When *StopURL* holds a value, the button is rendered as a link to that URL. The URL is sanitized and attribute encoded before it reaches the markup. When *StopURL* is empty, the button is rendered as a submit button inside a small **data-sgc-ws-send** form, and pressing it sends this payload over the WebSocket channel.

```
{ "action": "userStopImpersonate", "banner": "adminImpersonateBar" }
```

The action string is exactly **userStopImpersonate**, and the **banner** field carries the element id of the bar: the value of *BannerID* when that property is set, and the automatically generated id when it is left empty. Set *BannerID* explicitly whenever the server needs to recognise the bar by name. That is the contract between the bar and the server. Handle the incoming message on the server. This component publishes no event of its own for it, the message arrives through the WebSocket/HTMX engine like any other **data-sgc-ws-send** payload, and the server must check that the acting session really is impersonating before it ends anything.

Property	Type	Default	Description
StopURL	string	''	When set, the Stop button is a link to this URL. When empty, the button submits a data-sgc-ws-send form that sends the userStopImpersonate action.

Appearance

What it does: *Color* selects the background of the bar, *Sticky* decides whether the bar is pinned to the top of the viewport, and *BannerID* names the root element. Light backgrounds (hcLight, hcWarning, hcWhite) are given dark text, every other colour is given white text, so the caption stays readable whichever colour is chosen.

Property	Type	Default	Description
Color	TsgcHTMLColor	hcWarning	Background colour class of the bar. Text colour is chosen automatically for contrast.
Sticky	Boolean	True	True pins the bar with position:fixed;top:0;left:0;right:0;z-index:1055;. False lets the bar sit in the normal document flow.
BannerID	string	''	Id of the root div, also sent in the banner field of the payload. Empty means a unique id is generated on the first render.

BannerID is not hardcoded to a literal any more. Leave it empty and a unique element id is generated automatically the first time the component renders, and reused for the life of the instance, so two instances on the same page never collide and pushed fragments keep targeting the element this instance actually rendered. Set it explicitly if you need a stable, known id (for example to reach the element from your own JavaScript, or to recognise the bar in the payload). The generated id is **sgcImpersonateBanner** plus a process-unique number, such as **sgcImpersonateBanner4**.

Output

The inherited **HTML** property returns a single ** ** that holds the message text and either a link to *StopURL* or a small **data-sgc-ws-send** form with a submit button. There is no external CSS and no external JavaScript beyond the Bootstrap classes the markup references.

Reference

- Properties — 7 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events of its own.

TsgcHTMLComponent_RolesPermissions

TsgcHTMLComponent_RolesPermissions renders a role by permission checkbox matrix: permissions down the rows, roles across the columns, one checkbox per cell.

Introduction

TsgcHTMLComponent_RolesPermissions emits an HTML table where every permission is a row and every role is a column. The cell where a row and a column meet holds a checkbox, and the checkbox is ticked when that role has been granted that permission. Permissions can be grouped under category headers, and each role name is drawn as a coloured badge in the column header.

When the user ticks or unticks a checkbox the component sends a flat JSON payload over the WebSocket channel using `window.sgcHTMX.send`, and it also dispatches a DOM CustomEvent named `sgcRolesPermissions:grantChanged` so that other scripts on the page can react to the change.

Quick Start

1. Drop a **TsgcHTMLComponent_RolesPermissions** component on the form (or create it at runtime).

2. Add the roles with **AddRole**, add the permissions with **AddPermission**, and mark the granted pairs with **SetGrant**.

3. Read the **HTML** property and serve it (or assign it to a page template's BodyContent).

```
TsgcHTMLComponent_RolesPermissions *oMatrix =
  new TsgcHTMLComponent_RolesPermissions(NULL);
try
{
  oMatrix->AddRole("admin", "Administrator")->Color = hcDanger;
  oMatrix->AddRole("editor", "Editor");
  oMatrix->AddRole("viewer", "Viewer")->Color = hcSecondary;

  oMatrix->AddPermission("users.view", "View users", "Users");
  oMatrix->AddPermission("users.edit", "Edit users", "Users");
  oMatrix->AddPermission("reports.view", "View reports", "Reports");

  oMatrix->SetGrant("admin", "users.view", true);
  oMatrix->SetGrant("admin", "users.edit", true);
  oMatrix->SetGrant("editor", "users.view", true);
  oMatrix->SetGrant("viewer", "reports.view", true);

  html = oMatrix->HTML;
}
__finally
{
  delete oMatrix;
}
```

Security

These components render the user interface only. They do NOT enforce authorization.

Actions and grants arrive from the browser as client-supplied data, so a hostile client can forge any action, user id, role id or permission id, including ones that were never rendered, and including when **ReadOnly** is True (ReadOnly is a rendering hint, not a control).

The host application must verify on the server, for every message it receives, that the authenticated acting user is permitted to perform that action on that target, and it should audit impersonation, deletion and permission grants.

Hiding a button in the UI hides nothing from an attacker.

This matrix edits the very table that authorization decisions are based on, so it is the highest-value target in the application. **ReadOnly** only adds the `disabled` attribute to the rendered checkboxes. It does not stop a client sending a `grantChanged` message, and that message can name a roleId or a permissionId that was never rendered. Check every grant message against the acting user's own permissions on the server before you apply it, and log every accepted change.

Role check. Set *RequiredRole* to render the matrix, and to let **ProcessGrantChanged** do anything, only for a request session that has that role (the session set by the HTMX engine when it has an Auth). Any other request gets a danger alert with *AccessDeniedText*. The check covers the role only: every message still needs the server side authorization described above. See Sessions, CSRF and Authentication.

Property	Type	Default	Description
RequiredRole	string	''	Role the request session must have. Empty disables the check.
AccessDeniedText	string	'Access denied.'	Text of the alert shown instead of the matrix.

Roles and Permissions

What it does: *Roles* holds the columns and *Permissions* holds the rows. Add a role with **AddRole** and a permission with **AddPermission**. A **TsgcHTMLRole** exposes *Id*, *Name*, *Description* and *Color*, and the role name is rendered as a coloured badge in the column header. A **TsgcHTMLPermission** exposes *Id*, *Name*, *Description* and *Category*. *PermissionCaption* is the text of the top left header cell, and *ShowDescriptions* prints the role and permission descriptions under their names.

Property	Type	Default	Description
Roles	TsgcHTMLRoles	empty	Collection of TsgcHTMLRole items, one per matrix column.
Permissions	TsgcHTMLPermissions	empty	Collection of TsgcHTMLPermission items, one per matrix row.
PermissionCaption	string	'Permission'	Text of the first header cell, above the permission names.
ShowDescriptions	Boolean	True	Shows the role and permission descriptions under their names.

Role and permission names and descriptions are HTML-encoded when they are written to the page, and the ids only ever appear attribute-encoded in the `data-sgc-role` and `data-sgc-permission` attributes, never inside a

JavaScript string literal. That is output encoding, it protects the page against cross-site scripting. It is not authorization, and it says nothing about what the browser is allowed to send back.

Categories

What it does: Each permission has a *Category*, the group header it is listed under. When *ShowCategories* is True, the rows are grouped by Category and each distinct category opens one full-width group header row carrying the category name.

The order in which the permissions were added does not matter. The distinct categories are collected first, and each header is emitted exactly once, so a collection added in the order Billing, Users, Billing still renders a single Billing header with both Billing permissions under it. There is no need to sort the collection by category.

The grouping is a rendering one: the *Permissions* collection is never sorted or reordered, so the indices the caller holds stay valid. The categories keep the order in which they first appear in the collection, and inside a category the permissions keep their collection order. The match is exact and case sensitive, so `Billing` and `billing` are two different groups.

A permission with an empty Category is a group of its own, rendered without a group header. With *ShowCategories* False nothing is grouped and the rows are emitted in plain collection order.

Property	Type	Default	Description
ShowCategories	Boolean	True	Groups the rows by Category, one full-width group header row per distinct category.

Grants

What it does: A grant is a role id plus a permission id. Grants are kept internally in a sorted list of keys, and a pair that is present in the list is granted. **SetGrant** adds or removes a pair, **GetGrant** tests whether a pair is present, **ClearGrants** empties the list without touching the roles or the permissions, and **Clear** empties the roles, the permissions and the grants.

When the user changes a checkbox in the browser, the component sends this payload over the WebSocket channel through `window.sgcHTMX.send`, and dispatches it as the `detail` of the DOM CustomEvent `sgcRolesPermissions:grantChanged`.

```
{ action: "grantChanged", matrix: "sgcRolesTenantA",
  roleId: "admin", permissionId: "users.edit", granted: true }
```

The action string is exactly `grantChanged`, that is the contract between the browser and the server. *matrix* is the element id of the matrix, so a page can carry more than one matrix and the server can still tell them apart. It is the value of **MatrixID** when that property is set, and the automatically generated id when it is left empty, so set **MatrixID** explicitly whenever the server needs to recognise the matrix by name.

On the server, when a grant message arrives, first authorize it, then call **ProcessGrantChanged**. That method applies **SetGrant** and raises **OnGrantChanged**. The only check the component makes is *RequiredRole*, against the role of the request session, so the authorization of each grant has to happen before the call.

Bulk toggles: when *ReadOnly* is False, each column header carries a "select all permissions for this role" checkbox and each permission row carries a "select all roles for this permission" checkbox. Toggling one of them fires an

individual grant message for every cell it actually changes, so the server sees the same `grantChanged` payload it would see from a single click, once per changed cell.

Appearance

What it does: *Striped*, *Bordered* and *Compact* control the table style. *StickyHeader* wraps the table in a scroll area of height *MaxHeight* and pins the header row while the body scrolls, and *MaxHeight* is only used when StickyHeader is True. *EmptyText* is the text of the single centred row rendered when there are no permissions. *MatrixID* is the id of the generated table, and *ReadOnly* disables the checkboxes.

When *ReadOnly* is True every checkbox is rendered with the `disabled` attribute, the row and column bulk toggles are not rendered at all, and the component emits no script. That is a rendering decision. It does not prevent a client from sending a grant message, see the Security section above.

Property	Type	Default	Description
ReadOnly	Boolean	False	Renders every checkbox disabled, drops the bulk toggles and emits no script.
Striped	Boolean	True	Alternates the row background colour.
Bordered	Boolean	True	Draws borders around every cell.
Compact	Boolean	False	Reduces the cell padding.
StickyHeader	Boolean	False	Pins the header row and scrolls the body inside a MaxHeight scroll area.
MaxHeight	string	'60vh'	Height of the scroll area. Only used when StickyHeader is True.
EmptyText	string	'No permissions defined.'	Text shown in one centred row when Permissions is empty.
MatrixID	string	''	The id of the generated <table> element and the matrix name in the payload. Empty means a unique id is generated on the first render.

MatrixID is not hardcoded to a literal any more. Leave it empty and a unique element id is generated automatically the first time the component renders, and reused for the life of the instance, so two instances on the same page never collide and pushed fragments keep targeting the element this instance actually rendered. Set it explicitly if you need a stable, known id (for example to reach the element from your own JavaScript, or to recognise the matrix in the payload). The generated id is `sgcRolesPermissions` plus a process-unique number, such as `sgcRolesPermissions7`.

Data Binding

What it does: The three parts of the matrix are loaded separately. **LoadRolesFromDataSet** fills the columns, **LoadPermissionsFromDataSet** fills the rows, and **LoadGrantsFromDataSet** fills the ticked cells, one row per granted pair.

Assigning the inherited *DataSource* reloads the **grants only**. The bound dataset is expected to hold one row per granted role and permission pair, and the column names are taken from *GrantRoleField*, *GrantPermissionField* and *GrantGrantedField*. The roles and the permissions are never loaded from DataSource, they are always loaded explicitly with LoadRolesFromDataSet and LoadPermissionsFromDataSet. This asymmetry is deliberate, because roles and permissions usually live in different tables from the grants.

Property	Type	Default	Description
GrantRoleField	string	'role_id'	Name of the role id column read from DataSource.
GrantPermissionField	string	'permission_id'	Name of the permission id column read from DataSource.
GrantGrantedField	string	'granted'	Name of the boolean column read from DataSource. When the column is missing the row counts as granted.

```
oMatrix->LoadRolesFromDataSet(qryRoles, "id", "name", "description");
oMatrix->LoadPermissionsFromDataSet(qryPermissions, "id", "name", "description", "category");
oMatrix->LoadGrantsFromDataSet(qryGrants, "role_id", "permission_id", "granted");
```

Output

The inherited **HTML** property returns a responsive `<div>` wrapping the `<table>`, whose id is **MatrixID**. When **StickyHeader** is True the wrapper is also a scroll area of height **MaxHeight** and carries an inline `<style>` block that pins the header row, so the component renders correctly even when it is used outside a page builder.

When **ReadOnly** is False the markup is followed by an inline `<script>` that wires one change listener on the table. That listener sends a `grantChanged` payload for a cell checkbox, and for a row or column bulk toggle it sends one payload per cell it actually changes. When ReadOnly is True no script is emitted.

Serve the string directly, or assign it to the BodyContent of a page template to produce a full HTML document.

Reference

- Properties — 18 properties available on the component.
- Methods — 10 public methods.
- Events — 1 event, raised when an authorized grant change is applied.

TsgcHTMLEngine_Server

TsgcHTMLEngine_Server binds sgcHTML page rendering to a TsgcWSHTTPServer and serves rendered pages plus all embedded Bootstrap, htmx and Chart.js assets over HTTP.

Introduction

TsgcHTMLEngine_Server is the server-side glue between your sgcHTML pages and the underlying WebSocket HTTP server. Drop it on the form (or create at runtime), assign the Server property and call Start. The engine intercepts HTTP requests for the registered base URL, delegates them to your sgcHTML pages and automatically handles all /bootstrap.min.css, /bootstrap.bundle.min.js and /chart.umd.min.js requests from its embedded resources, so no CDN is required.

Quick Start

1. Drop a TsgcHTMLEngine_Server on the form (or create it at runtime).

2. Assign Server to your active TsgcWSHTTPServer.

3. Set BaseURL to the URL prefix for this engine (default '/').

4. Set Active to True.

```
TsgcHTMLEngine_Server *oEngine = new TsgcHTMLEngine_Server(NULL);
oEngine->Server  = WSSServer;
oEngine->BaseURL = "/";
oEngine->Active  = true;
```

Properties

What it does: Configures the binding between the engine and the HTTP server.

Property	Type	Default	Description
Server	TsgcWSHTTPServer	nil	The active WebSocket HTTP server to bind to.
Active	Boolean	False	Activates or deactivates the engine.
BaseURL	string	'/'	URL path prefix for all pages served by this engine.

Methods

- **RegisterPage(aPath: string; aPage: TObject)** — registers a page handler at the given sub-path.
- **HandleRequest(AContext, ARequest, AResponse)** — processes an incoming HTTP request; routes to page handlers or serves static assets.
- **Start / Stop** — activate or deactivate the engine (equivalent to setting Active).

Output

The engine serves fully rendered HTML pages and all required static assets (Bootstrap CSS/JS, htmx, Chart.js) directly from embedded resources. No external CDN or file-system deployment is needed.

Reference

- Properties — 3 properties available on the component.
- Methods — 3 public methods.
- Events — this component has no published events.

TsgcHTMX_Engine_Server

TsgcHTMX_Engine_Server handles htmx partial-page swap requests over HTTP and broadcasts live out-of-band fragments to all connected browsers over WebSocket.

Introduction

TsgcHTMX_Engine_Server extends the base HTML engine with htmx support. Assign a TsgcWSHTTPServer to Server and the engine hooks it: inbound HTTP requests (GET, HEAD and POST, the ones Indy delivers to OnCommandGet) are routed through DispatchRequest (the page, the built-in assets and the router routes), and inbound WebSocket messages are routed through DispatchMessage, which raises OnHTMXMessage so your handler can return the HTML fragment to send back. For server-initiated updates, call BroadcastFragment (all browsers) or PushFragment (one connection) to send an out-of-band htmx payload, with no browser-side polling.

The page template answers a boosted navigation with the title and the body content only, and **SSE** serves the live updates as an event stream when a WebSocket is not an option. See Navigation and Live Updates and WebSocket or SSE.

Assign a TsgcHTMLAuth to **Auth** to add cookie sessions, route gates (RequireLogin, RequireRoles), CSRF protection and built-in login and logout endpoints, see Sessions, CSRF and Authentication. **DispatchDirect** runs the same dispatch without any server or socket, see Routing and Testing sgcHTML Pages.

Quick Start

1. Create a TsgcHTMX_Engine_Server and assign Server.

2. Handle OnHTMXMessage to return fragment HTML for each request.

3. Call BroadcastFragment from any thread to push live DOM updates.

```
TsgcHTMX_Engine_Server *oHTMX = new TsgcHTMX_Engine_Server(NULL);
oHTMX->Server        = WSSServer;
oHTMX->OnHTMXMessage = HandleHTMXMessage;
oHTMX->BroadcastFragment(
  "<div id=\"kpi-revenue\" hx-swap-oob=\"true\">" +
  CurrToStr(FRevenue) + "</div>");
```

Properties

What it does: Binds the engine to a server and, optionally, a template and router.

Property	Type	Default	Description
Server	TsgcWSHTTPServer	nil	The WebSocket HTTP server this engine hooks into.

Tem-plate	TsgcHTMLTemplate_HTMX	nil	Document template used to wrap routed page HTML.
Router	TsgcHTMX_Router	nil	Router that maps htmx request paths to fragment handlers.
Auth	TsgcHTMLAuth	nil	Optional authentication: sessions, route gates, CSRF protection and the login and logout end-points.
SSE	TsgcHTMXSSE_Options	off	Server-sent events transport: the engine serves an event stream as an alternative to the WebSocket.

Events

Event	Type	Description
OnHT-MXMessage	TsgcHTMXMes-sageEvent	Fired for each inbound htmx WebSocket message; set aResponse to the HTML to send back.

Methods

- **DispatchMessage(aConnection, aText): string** — processes an inbound WebSocket message, fires On-HTMXMessage and returns the response.
- **DispatchRequest(AContext, ARequestInfo, AResponseInfo): Boolean** — handles an inbound Indy HTTP request: page, assets, routes and, with Auth, the login and logout endpoints.
- **DispatchDirect(aRequest, aResponse): Boolean**: serves a transport-neutral TsgcHTMXDispatchRequest into a TsgcHTMXDispatchResponse, with no server involved.
- **CanDispatchRequest(aDocument[, aMethod]): Boolean**: tells whether the engine would serve a path, for GET or for the given method.
- **BroadcastFragment(aHTML)** — pushes an out-of-band htmx payload to every connected WebSocket client.
- **PushFragment(aGuid, aHTML)** — pushes an out-of-band htmx payload to a single connection by GUID.
- **GetPageHTML: string** — returns the full page HTML assembled from the Template and Router.
- **GetBoostedPageHTML: string**: returns the answer of a boosted navigation, the title of the template plus the inner HTML of its body.

PUT, PATCH and DELETE

The engine hooks the OnCommandGet event of Server, which Indy fires for GET, HEAD and POST. PUT, PATCH and DELETE arrive in OnCommandOther, which the engine does not hook. Forward that event to DispatchRequest to reach routes whose Methods include those verbs:

```
WSSServer->OnCommandOther = ServerCommandOther;

void __fastcall TForm1::ServerCommandOther(TIdContext* AContext,
```

```
  TIdHTTPRequestInfo* ARequestInfo, TIdHTTPResponseInfo* AResponseInfo)
{
  oHTMX->DispatchRequest(AContext, ARequestInfo, AResponseInfo);
}
```

Output

BroadcastFragment sends the supplied HTML string wrapped as an htmx out-of-band swap payload over the WebSocket connection to every connected browser. The browser's htmx script processes the hx-swap-oob attribute and replaces the matching DOM element without a full page reload.

Reference

- Properties — 5 properties available on the component.
- Methods — 8 public methods.
- Events — 1 published event.

TsgcHTMX_Engine_Server_WebBrokerBridge

TsgcHTMX_Engine_Server_WebBrokerBridge serves realtime sgcHTML pages from a DataSnap or WebBroker application hosted on TsgcWSHTTPWebBrokerBridgeServer, sharing a single port with the DataSnap REST dispatcher.

Introduction

TsgcHTMX_Engine_Server_WebBrokerBridge descends from TsgcHTMX_Engine_Server and is designed for DataSnap and WebBroker applications hosted on TsgcWSHTTPWebBrokerBridgeServer, the drop-in replacement for TIdHTTPWebBrokerBridge. Assign the bridge server to the Server property and the engine claims only the requests it can serve. Every other request falls through untouched to the WebBroker or DataSnap dispatcher, so the /datasnap/* REST methods keep working on the same port. If your application already has its own OnCommandRequest handler, the engine chains it and your handler runs first.

The Server property also accepts TsgcWSHTTP2WebBrokerBridgeServer, the HTTP/2 bridge server. This support requires the HTTP/2 feature (SGC_HTTP2 define). The behavior is the same over HTTP/1.1 and real HTTP/2 streams: the generated page, the built-in assets and the Router fragments are served by the engine, and every other request falls through to the WebBroker or DataSnap dispatcher. If the application is hosted on the HTTP.sys server, use TsgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge instead.

The unit is **only located in the Source folder**, it cannot be included in the sgcWebSockets packages because it links the WebBroker unit, which Delphi does not allow inside a package, so the engine must be created at runtime. Add the following unit to your project or set your search path to the Source folder:

- sgcHTMX_Engine_Server_WebBrokerBridge

The engine is part of the HTML pack, it is compiled when the SGC_HTMX define is enabled and supports Delphi 7 to Delphi 13.

Quick Start

1. Create a TsgcHTMX_Engine_Server_WebBrokerBridge in code and assign your TsgcWSHTTPWebBrokerBridgeServer (or TsgcWSHTTP2WebBrokerBridgeServer) to Server.

2. Assign a Router with your fragment handlers and build the page with the sgcHTML components or the node layer, then assign it to Template.

3. Call BroadcastFragment or PushFragment from any thread to push live DOM updates to the connected browsers.

Which Requests the Engine Claims

The engine intercepts the **GET** requests it can actually serve, and requests with other methods only when a router route answers that path with that method, or when they are a POST to the login or logout path of Auth (see CanDispatchRequest). Everything else continues to the WebBroker or DataSnap dispatcher.

Request	Handled by

GET / (the generated page)	Engine
GET built-in js and css assets	Engine
GET manifest.json and sw.js (when Template.PWAEnabled)	Engine
GET paths registered in the Router (routes with an empty Methods set or with hrmGet)	Engine
/datasnap/* REST methods	DataSnap dispatcher
POST, PUT, PATCH or DELETE to a Router route whose Methods include that method	Engine
POST to Auth.LoginPath or Auth.LogoutPath (when Auth is assigned)	Engine
Any other request (other methods, unregistered paths...)	WebBroker / DataSnap dispatcher

Any OnCommandRequest handler you assigned to the server before wiring the engine keeps working: the engine chains it and runs it first, so requests you mark as handled are never touched by the engine.

Realtime Updates

The full realtime support of TsgcHTMX_Engine_Server is inherited. Template.WSEndpoint (default '/htmx-ws') makes the generated page open a WebSocket connection back to the same port, and the interactive sgcHTML components (Grid, Chat, LogViewer, forms) work over that WebSocket channel. Call PushFragment to push a live HTML fragment to a single browser, or BroadcastFragment to push it to all connected browsers, with no browser-side polling.

Properties

Property	Type	Description
Server	TsgcWSHTTPServer	Assign your TsgcWSHTTPWebBrokerBridgeServer or TsgcWSHTTP2WebBrokerBridgeServer here. The engine hooks the server OnCommandRequest event and chains any handler already assigned.
Template	TsgcHTMLTemplate_HTMX	Document template used to build the page served at the root endpoint.
Router	TsgcHTMX_Router	Router that maps htmx request paths to fragment handlers.
Auth	TsgcHTMLAuth	Sessions, route gates, CSRF protection and the login and logout endpoints, see Sessions, CSRF and Authentication.

The methods (BroadcastFragment, PushFragment, GetPageHTML...) and the OnHTMXMessage event are inherited from TsgcHTMX_Engine_Server, refer to that topic for the full reference.

Demo

Find a full example in the folder **Demos\40.DataSnap\Server_Indy_HTTP_HTML**: a DataSnap ReverseString server plus a realtime sgcHTML dashboard, both served on a single port.

See Also

- sgcHTML pages on DataSnap Servers
- TsgcHTMX_Engine_Server
- TsgcHTMX_Engine_Server_WebBroker
- TsgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge
- TsgcWSHTTPWebBrokerBridgeServer
- TsgcWSHTTP2WebBrokerBridgeServer

TsgcHTMX_Engine_Server_WebBroker

TsgcHTMX_Engine_Server_WebBroker serves sgcHTML pages from any WebBroker application (standalone, ISAPI, Apache, CGI) through the standard IWebDispatch interface. No sgc server is required.

Introduction

TsgcHTMX_Engine_Server_WebBroker descends from TsgcHTMLEngine and implements the WebBroker **IWebDispatch** interface, so it plugs into any WebBroker application: a standalone TIdHTTPWebBrokerBridge executable, an ISAPI dll, an Apache module or a CGI executable. It serves the page built with the sgcHTML components at the root endpoint, the built-in js and css assets, the PWA manifest and service worker when enabled, and the htmx fragment routes registered in the Router. Paths it does not own are left untouched so other dispatchers, for example TDSHTTPWebDispatcher, continue processing the request.

No WebSocket Push under WebBroker

WebSocket fragment push is NOT available under plain WebBroker hosting. In the WebBroker model (and in particular under ISAPI, Apache and CGI) the request/response lifecycle owns the socket, so the engine cannot keep a WebSocket connection open to the browser. The constructor clears Template.WSEndpoint, so the generated page does not try to open a WebSocket and the interactive components degrade automatically to standard HTMX HTTP round-trips.

For periodic updates, use HTMX polling on the elements that must refresh themselves:

```
<div hx-get="/fragments/status" hx-trigger="every 5s" hx-swap="innerHTML"></div>
```

If you need true realtime push (PushFragment, BroadcastFragment, live Grid, Chat or LogViewer updates), host your WebModule on TsgcWSHTTPWebBrokerBridgeServer and use TsgcHTMX_Engine_Server_WebBrokerBridge instead.

Configuration

The unit is **included in the sgcWebSockets Delphi packages**. There is no palette registration, so the engine must be created at runtime. In C++Builder the unit is only located in the Source folder, add it to your project or set your search path to the Source folder:

- sgcHTMX_Engine_Server_WebBroker

The engine is part of the HTML pack, it is compiled when the SGC_HTMX define is enabled and supports Delphi 7 to Delphi 13.

Wiring Options

Option A: automatic dispatch through IWebDispatch. Create the engine with the TWebModule as Owner and the WebBroker dispatcher invokes it automatically. The PathMask property (default '*') controls which request paths are offered to the engine.

Option B: manual dispatch from a TWebActionItem handler. Call the public DispatchRequest method yourself. It returns False for paths the engine does not own, so other dispatchers continue.

Which Requests the Engine Handles

Request	Handled by
/ (the generated page, when a Template is assigned)	Engine
Built-in js and css assets	Engine
manifest.json and sw.js (when Template.PWAEnabled)	Engine
Paths registered in the Router (query and content fields are merged into the parameter list). A route with an empty Methods set answers every HTTP method here, except the POST of the Auth login and logout paths	Engine
POST to Auth.LoginPath and Auth.LogoutPath, when Auth is assigned, unless a route for that path lists hrmPost in its Methods	Engine
Any other path	DispatchRequest returns False, other dispatchers continue

Properties

Property	Type	Default	Description
Template	TsgcHTMLTemplate_HTMX	owned	Document template used to build the page served at the root endpoint. WSEndpoint is cleared by the constructor because there is no WebSocket under WebBroker.
Router	TsgcHTMX_Router	nil	Router that maps htmx request paths to fragment handlers.
Auth	TsgcHTMLAuth	nil	Sessions, route gates, CSRF protection and the login and logout endpoints, as described for

			TsgcHTMX_Engine_Server. Session and remember-me cookies are written to the WebBroker response.
En-abled	Boolean	True	When False, the IWebDispatch dispatch is disabled.
Path-Mask	string	'*'	Mask matched against the request path before the WebBroker dispatcher offers the request to the engine.

With **Auth** assigned, WebBroker hands the engine a fixed set of request headers (Accept, Accept-Language, Authorization, Content-Type, Host, Origin, Referer, User-Agent, the htmx headers and X-CSRF-Token) plus the Cookie header and the header named by Auth.CSRFHeaderName. A route with an empty Methods set never takes the POST of Auth.LoginPath or Auth.LogoutPath away from the built-in endpoints; only a route whose Methods include hrmPost does. See Sessions, CSRF and Authentication.

Methods

- **DispatchRequest(Request, Response): Boolean**. Processes a WebBroker request. Returns True when the engine handled it, and False for paths it does not own so other dispatchers continue.
- **GetPageHTML: string**. Returns the full page HTML assembled from the Template.

Demo

Find a full example in the folder **Demos\60.HTML\11.WebBroker**, with standalone and ISAPI targets.

See Also

- sgcHTML pages on DataSnap Servers
- TsgcHTMX_Engine_Server_WebBrokerBridge
- TsgcHTMX_Engine_Server

TsgcHTMX_Engine_Server_HTTPAPI_WebBroker

TsgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge serves realtime sgcHTML pages from a DataSnap or WebBroker application hosted on TsgcWSServer_HTTPAPI_WebBrokerBridge, the HTTP.sys bridge server, sharing a single port with the DataSnap REST dispatcher.

Introduction

TsgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge descends from TsgcHTMX_Engine_Server_HTTPAPI and is designed for DataSnap and WebBroker applications hosted on TsgcWSServer_HTTPAPI_WebBrokerBridge, the bridge server built on the Windows HTTP.sys kernel driver. Assign the bridge server to the Server property and the engine claims only the requests it can serve. Every other request falls through untouched to the WebBroker or DataSnap dispatcher, so the /datasnap/* REST methods keep working on the same port. If your application already has its own OnCommandRequest handler, the engine chains it and your handler runs first. The behavior is identical to the Indy engine TsgcHTMX_Engine_Server_WebBrokerBridge.

The unit is **included in the sgcWebSockets Delphi packages from Delphi 10.1 Berlin onwards**. There is no palette registration, so the engine must be created at runtime. In earlier Delphi versions and in C++Builder the unit is only located in the Source folder, add it to your project or set your search path to the Source folder:

- sgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge

The engine is part of the HTML pack, it is compiled when the SGC_HTMX define is enabled together with the HTTPAPI feature (SGC_HTTPAPI define). The HTTP.sys server is Windows only.

Quick Start

1. Create a TsgcHTMX_Engine_Server_HTTPAPI_WebBrokerBridge in code and assign your TsgcWSServer_HTTPAPI_WebBrokerBridge to Server.

2. Assign a Router with your fragment handlers and build the page with the sgcHTML components or the node layer, then assign it to Template.

3. Call BroadcastFragment or PushFragment from any thread to push live DOM updates to the connected browsers.

Which Requests the Engine Claims

The engine intercepts the **GET** requests it can actually serve, and requests with other methods only when a router route answers that path with that method, or when they are a POST to the login or logout path of Auth (see CanDispatchRequest). Everything else continues to the WebBroker or DataSnap dispatcher.

Request	Handled by
GET / (the generated page)	Engine
GET built-in js and css assets	Engine

GET manifest.json and sw.js (when Template.PWAEnabled)	Engine
GET paths registered in the Router (routes with an empty Methods set or with hrmGet)	Engine
/datasnap/* REST methods	DataSnap dispatcher
POST, PUT, PATCH or DELETE to a Router route whose Methods include that method	Engine
POST to Auth.LoginPath or Auth.LogoutPath (when Auth is assigned)	Engine
Any other request (other methods, unregistered paths...)	WebBroker / DataSnap dispatcher

Any OnCommandRequest handler you assigned to the server before wiring the engine keeps working: the engine chains it and runs it first, so requests you mark as handled are never touched by the engine.

Response cookies are supported on the HTTP.sys bridge, so WebBroker session cookies work as expected.

Realtime Updates

Template.WSEndpoint (default '/htmx-ws') makes the generated page open a WebSocket connection back to the same port, and the interactive sgcHTML components (Grid, Chat, LogViewer, forms) work over that WebSocket channel. Call PushFragment to push a live HTML fragment to a single browser, or BroadcastFragment to push it to all connected browsers, with no browser-side polling.

Properties

Property	Type	Description
Server	TsgcWSServer_HTTPAPI	Assign your TsgcWSServer_HTTPAPI_WebBrokerBridge here. The engine hooks the server OnCommandRequest event and chains any handler already assigned.
Template	TsgcHTMLTemplate_HTMX	Document template used to build the page served at the root endpoint.
Router	TsgcHTMX_Router	Router that maps htmx request paths to fragment handlers.
Auth	TsgcHTMLAuth	Sessions, route gates, CSRF protection and the login and logout endpoints, see Sessions, CSRF and Authentication.

The methods (BroadcastFragment, PushFragment, GetPageHTML...) and the OnHTMXMessage event are inherited from TsgcHTMX_Engine_Server_HTTPAPI and mirror the TsgcHTMX_Engine_Server API, refer to that topic for the full reference.

Plain HTTP.sys Servers

The parent engine TsgcHTMX_Engine_Server_HTTPAPI also serves the page, the built-in assets, the PWA files and the Router GET routes on a plain TsgcWSServer_HTTPAPI through its OnHTTPRequest event, no WebBroker involved. Use the WebBrokerBridge descendant described in this topic only when the application hosts a WebBroker or DataSnap module.

Demo

Find the DataSnap examples in the folder **Demos\40.DataSnap**: the HTTPAPI server variant hosts the DataSnap ReverseString server on the HTTP.sys bridge, and the Quick Start wiring above serves the realtime sgcHTML dashboard on top of it.

See Also

- sgcHTML pages on DataSnap Servers
- TsgcHTMX_Engine_Server
- TsgcHTMX_Engine_Server_WebBrokerBridge
- TsgcWSServer_HTTPAPI_WebBrokerBridge
- TsgcWebSocketServer_HTTPAPI

TsgcHTMLDesignServer

TsgcHTMLDesignServer binds a TsgcHTMLDesignSurface to a sgcWebSockets HTTP server, so the sgcHTML components dropped on a Delphi form are served as a web page without any request handling code.

Introduction

Drop a TsgcHTMLDesignServer on a form, assign **Server** and **Surface**, and the component hooks the **OnCommandGet** event of the server. The hook chains: an OnCommandGet handler you assigned yourself runs first and always wins, and the DesignServer only answers a request that your handler left without a response. This is the same chaining the TsgcHTMX_Engine_Server uses.

The DesignServer is the component behind the QuickStart demos. To build the same page in code instead, see Runtime vs Design-Time.

Zero-Code Mode (PagePath)

While **PagePath** is set (it is '/' by default), the DesignServer serves a complete page with no code on your side:

- When the Surface has no design tree of its own, the page is built automatically from the sgcHTML controls of the form that owns the Surface. The build runs when the server is hooked, while the form loads on the main thread, so no request ever sees a half built page.
- A request for PagePath returns the rendered page. The scripts and stylesheets the page links (Bootstrap, htmx and Chart.js) are served from the resources compiled into the executable, so no other handler and no files on disk are needed.
- Page rendering and the action handlers are serialized by one lock, so a handler that changes a component never runs while the page that reads it is being rendered.
- A path that no handler served gets **404 Not Found**, instead of the empty "200 OK" page Indy would send otherwise.
- Several DesignServers can share one HTTP server, each with its own PagePath, which turns several forms into one web site. The **Demos\60.HTML\00.QuickStart\06.Pages** demo serves '/', '/customers' and '/about' this way. The 404 is decided by the hook installed last, after your handler and every other DesignServer had their turn.

Set PagePath to an empty string to stop serving the page.

Quick Start

1. Drop a TsgcWebSocketHTTPServer, a TsgcHTMLDesignServer and a TsgcHTMLDesignSurface on a form.

2. Set Server and Surface of the DesignServer, and leave PagePath as '/'.

3. Drop the sgcHTML components of the page on the same form and start the server.

This is the whole code of the **Demos\60.HTML\00.QuickStart\01.HelloWorld** demo. Everything else is set in the Object Inspector.

Properties

Property	Type	Default	Description
Server	TsgcWSHTTPServer	nil	The HTTP server the DesignServer hooks, for example a TsgcWebSocketHTTPServer. Editions without the WebSocket server bind to TsgcHTTPServer instead.
Surface	TsgcHTMLDesignSurface	nil	The page: its title, its theme, its node tree and its **Actions**.
PagePath	string	'/'	The URL where the designed page is served. Empty disables it.
Enabled	Boolean	False	Turns on the designer routes used by Design in Browser. Keep it False in production: the page keeps being served and the designer routes stop answering.
BasePath	string	'/__sgcdesign'	The path under which the designer routes answer while Enabled is True.
ExportFolder	string	empty	The folder an export writes to, see Designing in the Browser. While it is empty an export writes nothing.
SourceUnitName	string	empty	The name of the Pascal unit an export writes.
Generator	TsgcHTMLDesignAIGenerator	nil	The AI page generator called by the designer, see AI Page Generator.

Events

Event	Type	Description
OnChange	TNotifyEvent	Fired after the designer changes the design.

The events of the page itself are the **OnExecute** handlers of the Surface actions, described below.

Methods

- **DispatchRequest(ARequestInfo, AResponseInfo): Boolean**: answers one Indy request the same way the OnCommandGet hook does (actions, page, assets and designer routes), and returns True when it filled the response.

Actions: Browser Events Run Delphi Code

Surface.Actions is a collection of **TsgcHTMLDesignAction** items, edited in the Object Inspector like any other collection. Each action says which component the user acts on, what the user does, and which component is redrawn afterwards.

Member	Type	Description
Source	TsgcHTMLComponent	The component whose markup carries the trigger, for example a button.
Target	TsgcHTMLComponent	The component that is rendered again, in place, after OnExecute.
Trigger	TsgcHTMLDesignTrigger	What the user does in the browser: dtClick (the default) for a click, dtSubmit for a form submission.
Kind	TsgcHTMLDesignActionKind	What the page does about it. The default, dakRefresh, redraws the Target.
Confirm	string	When set, the browser asks this question before running the action.
OnExecute	TsgcHTMLDesignActionEvent	The Delphi code of the action: `procedure(Sender: TObject; aAction: TsgcHTMLDesignAction; var aResponse: string)`. Leave aResponse empty and the Target is rendered again and swapped into the page in place, without a reload.
Params	TStrings	Public, read only. The values the browser sent with the request (form fields and query string), filled only while OnExecute runs.

A dtSubmit action whose Source is a TsgcHTMLComponent_Form follows the **Method** of the form: a POST form posts its fields, a GET form sends them in the query string. Either way they arrive in aAction.Params.

The action items of the **Demos\60.HTML\00.QuickStart\03.Events** demo, as they are stored in the form:

And the handler of the first action. Clicking "Say hello" in the browser runs it on the server, and ResultPanel is redrawn with the new text:

Threads and User Input

OnExecute runs on a server thread, not on the main thread, so it must not touch the VCL controls of the desktop window. The sgcHTML components are safe to change, because the server runs one action at a time and never while the page is being rendered. While Enabled is True for Design in Browser, the designer marshals the handler to the main thread instead.

A value that came from the browser is user input. Encode it with **sgcHTMLEncode** (unit sgcHTMX_Helpers) before you put it into a property that is rendered as HTML.

Design in Browser

The DesignServer is also a design host. With **Enabled** set to True, its designer routes under BasePath let you edit the page live in a browser, against the data the application is already serving, and the edits are written back to the form. The canvas, the Web Design tab of the IDE, the local listener and exporting are described in Designing in the Browser. The designer routes are not authenticated, so Enabled is False by default and must stay False on a production server.

Demo

Every demo in **Demos\60.HTML\00.QuickStart** is built on a TsgcHTMLDesignServer, see QuickStart.

See Also

- QuickStart
- Runtime vs Design-Time
- Designing in the Browser
- TsgcHTMX_Engine_Server
- TsgcHTMLComponent_Form

TsgcHTMLPageBuilder

TsgcHTMLPageBuilder assembles registered sgcHTML components into a complete Bootstrap 5 page, with a design-time visual editor in the Delphi IDE.

Introduction

TsgcHTMLPageBuilder is a design-time and runtime page assembly tool. Drop components onto its canvas in the IDE or register them programmatically with RegisterComponent; then read the HTML property (or call GetHTML) to get the fully composed page. The builder wraps the assembled fragment in a Bootstrap 5 document, applies the active theme and optional sidebar or navigation bar, and returns a complete, self-contained HTML string ready to serve. It can also emit a web app manifest and service worker when PWAEnabled is True.

Quick Start

1. Create a TsgcHTMLPageBuilder.

2. Set Title, HtmlLang and Theme.

3. Call RegisterComponent for each sgcHTML component in display order.

4. Read the HTML property to produce the complete page.

```
TsgcHTMLPageBuilder *oBuilder = new TsgcHTMLPageBuilder(NULL);
try
{
  oBuilder->Title    = "Sales Dashboard";
  oBuilder->HtmlLang = "en";
  oBuilder->Theme    = ptDark;
  oBuilder->RegisterComponent(oNavBar);
  oBuilder->RegisterComponent(oChart);
  String html = oBuilder->HTML;
}
__finally { delete oBuilder; }
```

Properties

What it does: Controls the page-level settings, the optional sidebar or navigation bar and the PWA output.

Every one of these properties is a plain value: the title, the brands, the section titles and the footer text are encoded as text, SidebarWidth is filtered as a CSS value, and the PWA names, colours and icon paths are escaped for the JSON of the manifest.

Property	Type	Default	Description
Title	string	''	Page title placed in the document <title> tag.

HtmlLang	string	'en'	lang attribute on the <html> element of the generated page.
Theme	TsgcHTML-PageTheme	ptDefault	Page theme: ptDefault, ptDark or ptDarkPro.
DarkMode	Boolean	False	Switches the page to the built-in dark styling.
PageTemplate	TsgcHTML-PageTemplate	tpCustom	Preset layout (dashboard, admin panel, login page and more).
ShowSidebar / ShowNavBar	Boolean	False	Render a sidebar and/or a top navigation bar.
IncludeBootstrap	Boolean	True	Links Bootstrap in the generated page when True.
PWAEnabled	Boolean	False	Emits a manifest and service worker so the page installs as a PWA.

See the Properties page for the full list, including the sidebar, navigation-bar and PWA settings.

Methods

- **RegisterComponent(aComponent)** — adds an sgcHTML component to the page in display order.
- **UnRegisterComponent(aComponent)** — removes a component from the page.
- **GetHTML: string** — renders all components and returns the complete page (also the read-only HTML property).
- **PreviewInBrowser** — opens the page in the default browser for a quick design-time preview.

Output

GetHTML concatenates the HTML of every registered component (in registration order), wraps the result in a Bootstrap 5 document with the active theme and any sidebar or navigation bar, and returns a complete, self-contained HTML page.

Reference

- Properties — page, navigation and PWA settings.
- Methods — public methods.
- Events — this component has no published events.

TsgcHTMLTemplate_Bootstrap

TsgcHTMLTemplate_Bootstrap wraps any HTML body content in a complete, self-contained Bootstrap 5 document with configurable theme, language and text direction.

Introduction

TsgcHTMLTemplate_Bootstrap is the document-level wrapper for sgcHTML. Assign your rendered component HTML to BodyContent and read the inherited HTML property to get a complete, self-contained Bootstrap 5 page. The template injects the Bootstrap CSS/JS link tags (from BootstrapCSSPath and BootstrapJSPath), sets the html lang and dir attributes from HtmlLang and HtmlDir, and applies the active theme through Bootstrap's data-bs-theme attribute from HtmlTheme.

Use CustomHead to inject custom meta tags, links or scripts into the document head, and CustomCSS to inject an inline <style> block. For a full, branded palette assign a TsgcHTMLThemeBuilder to the ThemeBuilder property; its generated CSS (and, for tmSystem, its operating-system sync script) is injected automatically.

Quick Start

1. Create a TsgcHTMLTemplate_Bootstrap.

2. Set Title, HtmlTheme and HtmlLang.

3. Assign BodyContent to the concatenated HTML of your components.

4. Read the HTML property.

```
TsgcHTMLTemplate_Bootstrap *oTemplate = new TsgcHTMLTemplate_Bootstrap(NULL);
try
{
  oTemplate->Title       = "My App";
  oTemplate->HtmlTheme   = "dark";
  oTemplate->HtmlLang    = "en";
  oTemplate->BodyContent = oNavBar->HTML + oGrid->HTML;
  Response->ContentText  = oTemplate->HTML;
}
__finally { delete oTemplate; }
```

Document Settings

What it does: Controls the structure, language and metadata of the generated HTML document.

Property	Type	Default	Description
Title	string	'sgcWebSockets'	Text for the <title> element.
HtmlLang	string	'' ('en')	lang attribute on the <html> element. Falls back to 'en' when empty.

HtmlTheme	string	''	When set, emitted as the data-bs-theme attribute on <html> (for example 'light' or 'dark').
HtmlDir	string	''	When set, emitted as the dir attribute on <html> (for example 'ltr' or 'rtl').
Charset	string	'utf-8'	Value of the <meta charset> tag.
Viewport	string	'width=device-width, initial-scale=1'	Content of the viewport meta tag; empty omits it.
BodyContent	string	''	HTML fragment placed inside the document <body>.
BodyClass	string	''	class attribute added to the <body> element.
Page	TsgcHTMLPage	(node page)	Node-based body; its nodes render instead of BodyContent when present.

Assets, Theme and Styling

What it does: Controls the Bootstrap assets and the colour theme of the page.

Property	Type	Default	Description
IncludeCSS	Boolean	True	Links the Bootstrap stylesheet in the head when True.
IncludeJS	Boolean	True	Adds the Bootstrap bundle script before </body> when True.
BootstrapCSSPath	string	'/bootstrap.min.css'	URL of the Bootstrap stylesheet linked when IncludeCSS is True.
BootstrapJSPath	string	'/bootstrap.bundle.min.js'	URL of the Bootstrap bundle script added when IncludeJS is True.
ThemeBuilder	TsgcHTMLThemeBuilder	nil	When assigned, its generated CSS (and OS sync script for tmSystem) is injected into the head.
ThemeCSS	string	''	Raw CSS emitted as a final <style> override.
CustomCSS	string	''	Inline CSS injected as a <style> block in the head.

CustomHead	string	''	Raw markup appended to the <head> before it closes.
DarkMode	Boolean	False	Injects a built-in dark colour override when True. Prefer ThemeBuilder for full palettes.

Navigation

What it does: These eight options turn the page into a single page application, with no JavaScript of your own. Every one of them is off by default and, while all of them are off, the page is rendered exactly as before. With any of them on, the head loads htmx, the extensions that option needs and sgcHTMX.min.js from the root of the site. Navigation and Live Updates describes them in detail.

Property	Type	Default	Description
Boost	Boolean	False	Loads links and forms with htmx and swaps them into the body (hx-boost).
PushURL	Boolean	False	Pushes the URL of every htmx request into the browser history (hx-push-url).
DefaultSwap	string	''	Default swap style of the page. A value that starts with morph loads the idiomorph extension.
ViewTransitions	Boolean	False	Animates the swaps and the page navigations with the View Transitions API.
LoadingIndicator	Boolean	False	Thin progress bar at the top of the page while a request runs.
DisableOnSubmit	Boolean	False	Disables the submit buttons of a form while its request runs.
ScrollRestoration	Boolean	False	Remembers and restores the scroll position of every page.
ErrorToast	Boolean	False	Toast shown when a request fails.

Output

The HTML property (read-only, inherited) assembles the complete Bootstrap 5 document and returns it as a string. It can be assigned directly to the HTTP response body.

Reference

- Properties — 26 properties available on the component.
- Methods — the read-only HTML property that returns the document.
- Events — this component has no published events.

TsgcHTMLTemplate_HTMX

TsgcHTMLTemplate_HTMX is the page template of TsgcHTMX_Engine_Server: a Bootstrap document that loads htmx, opens the realtime channel of the page and can show a reconnect dialog.

Introduction

TsgcHTMLTemplate_HTMX is the page template of TsgcHTMX_Engine_Server. It descends from TsgcHTMLTemplate_Bootstrap, so it brings the same head, theme, PWA and navigation options, and adds the realtime transport of the page: the htmx scripts, the WebSocket or SSE attributes of the body, and the optional reconnect dialog.

The engine owns one instance, reachable through its **Template** property, and renders it for every request of the page. The unit is **sgcHTMX_Template**.

Quick Start

1. Set the page content and the head options on the template of the engine, as with TsgcHTMLTemplate_Bootstrap.

2. Choose the realtime transport with **Transport**, and the endpoints with WSEndpoint or with the SSE options of the engine.

3. Turn on the navigation options you want, and **ReconnectUI** to tell the user when the connection is down.

```
oHTMX->Template->Title = "Dashboard";
oHTMX->Template->BodyContent = oPage->HTML;
oHTMX->Template->WSEndpoint = "/htmx-ws";
oHTMX->Template->Transport = htAuto;
oHTMX->Template->ReconnectUI = true;
oHTMX->Template->Boost = true;
oHTMX->Template->DefaultSwap = "morph:innerHTML";
```

Realtime Transport

What it does: *Transport* picks the channel of the live updates, *WSEndpoint* is the WebSocket path and the three SSE properties are the paths and the event name of the event stream. *ReconnectUI* renders the dialog shown while the channel is down.

Property	Type	Default	Description
Transport	TsgcHTMXTransport	htWebSocket	htWebSocket, htSSE or htAuto.
WSEndpoint	string	'/htmx-ws'	Path of the WebSocket (ws-connect).
ReconnectUI	Boolean	False	Renders the #sgc-reconnect dialog.

SSEEndpoint	string	'/htmx-sse'	Path of the event stream (sse-connect). Public, copied by the engine.
SSEPostEnd-point	string	'/htmx-sse/send'	Path of the upstream POST (data-sgc-sse-post). Public, copied by the engine.
SSEEvent-Name	string	'sgc'	Event name of the pushed fragments. Public, copied by the engine.
HTMXVer-sion	string	'2.0.4'	Informative version of htmx.

See WebSocket or SSE for the whole transport picture and Navigation and Live Updates for the navigation options.

Scripts the Page Loads

- `/htmx.min.js` and `/sgcHTMX.min.js`, always.
- `/htmx-ext-ws.min.js` with the WebSocket transport, `/htmx-ext-sse.min.js` with the SSE transport, and both with htAuto.
- `/sgcWebSockets.js` when WSEndpoint is not empty.
- `/idiomorph-ext.min.js` when DefaultSwap starts with morph, and `/htmx-ext-head-support.min.js` when Boost is on.
- `/bootstrap.min.css` and `/bootstrap.bundle.min.js` through the inherited IncludeCSS and IncludeJS.

The engine serves all of them; another host can serve them with `sgcHTMXGetAsset`.

See Also

- TsgcHTMX_Engine_Server
- TsgcHTMLTemplate_Bootstrap
- Navigation and Live Updates
- WebSocket or SSE

Reference

- Properties: 7 properties of this class, plus the inherited ones.
- Methods: 1 public method of this class.
- Events: this component has no published events.

TsgcHTMLThemeController

TsgcHTMLThemeController reads a cookie or query parameter to determine the active Bootstrap theme and injects the theme initialisation script into each page. Use together with TsgcHTMLThemeBuilder for custom brand colours.

Introduction

TsgcHTMLThemeController sits between the HTTP request and the page template. It reads the user's theme preference from a cookie or ?theme= query parameter (falling back to DefaultTheme when neither is present) and returns the correct Bootstrap theme initialisation inline script. Assign the script string to TsgcHTMLTemplate_Bootstrap.HeadContent so every page switches theme correctly without a round-trip.

Pair TsgcHTMLThemeController with TsgcHTMLThemeBuilder to overlay your brand colours on top of the Bootstrap colour system. See also TsgcHTMLThemeBuilder.

Quick Start

1. Create a TsgcHTMLThemeController.

2. Set CookieName and DefaultTheme.

3. In your request handler, call GetThemeScript(ARequest) and append the result to oTemplate.HeadContent.

```
TsgcHTMLThemeController *oTheme = new TsgcHTMLThemeController(NULL);
try
{
  oTheme->CookieName   = "app_theme";
  oTheme->DefaultTheme = htSystem;
  oTemplate->HeadContent += oTheme->GetThemeScript(ARequest);
}
__finally { delete oTheme; }
```

Theme Settings

What it does: Reads the user's theme preference and emits an inline initialisation script.

Property	Type	Default	Description
CookieName	string	'theme'	Name of the cookie storing the user's preference.
DefaultTheme	TsgcHTMLTheme	htSystem	Fallback when no cookie or query parameter is found.
QueryParam	string	'theme'	URL query parameter for per-request theme override.

Methods

- **GetThemeScript(aRequest): string** — returns an inline <script> element that sets Bootstrap's theme on page load.

Output

GetThemeScript returns a small inline <script> block. Inject it into TsgcHTMLTemplate_Bootstrap.HeadContent before reading the HTML property.

Reference

- Properties — 3 properties available on the component.
- Methods — 1 public method.
- Events — this component has no published events.

TsgcHTMLThemeBuilder

TsgcHTMLThemeBuilder generates a <style> block that overrides Bootstrap CSS custom properties with your brand colours, fonts and radius, with light, dark and system modes.

Introduction

TsgcHTMLThemeBuilder takes your brand palette and emits CSS that overrides Bootstrap CSS custom properties (such as --bs-primary, --bs-body-bg and --bs-border-color) so every Bootstrap component picks up your colours without writing any CSS manually. It also controls the body font stack and the corner radius.

The Mode property selects tmLight, tmDark or tmSystem. In tmSystem mode the builder emits a prefers-color-scheme media query with the dark palette plus a small script that keeps Bootstrap's data-bs-theme attribute in sync with the operating-system colour scheme. The SetLightDefaults, SetDarkDefaults and SetSystemDefaults helpers load a complete built-in palette in one call.

The simplest way to apply a theme is to assign the builder to TsgcHTMLTemplate_Bootstrap.ThemeBuilder; the template then injects the generated CSS (and the system script) into the document head for you. See also TsgcHTMLThemeController for per-request theme selection.

Quick Start

1. Create a TsgcHTMLThemeBuilder.

2. Set PrimaryColor and any other brand colours (or call a preset).

3. Assign it to TsgcHTMLTemplate_Bootstrap.ThemeBuilder.

```
TsgcHTMLThemeBuilder *oBuilder = new TsgcHTMLThemeBuilder(NULL);
oBuilder->PrimaryColor   = "#7C3AED";
oBuilder->SecondaryColor = "#1E293B";
oBuilder->Mode           = tmSystem;
oTemplate->ThemeBuilder  = oBuilder;
Response->ContentText    = oTemplate->HTML;
```

Palette

What it does: Maps brand colours, the font and the radius to Bootstrap CSS custom properties.

Property	Type	Default	Description
PrimaryColor	string	'#0d6efd'	Overrides --bs-primary.
SecondaryColor	string	'#6c757d'	Overrides --bs-secondary.
SuccessColor	string	'#198754'	Overrides --bs-success.

DangerColor	string	'#dc3545'	Overrides --bs-danger.
WarningColor	string	'#ffc107'	Overrides --bs-warning.
InfoColor	string	'#0dcaf0'	Overrides --bs-info.
BodyBackground	string	'#ffffff'	Page background (--bs-body-bg).
BodyColor	string	'#212529'	Default text colour (--bs-body-color).
CardBackground	string	'#ffffff'	Cards, list groups, modals and controls.
BorderColor	string	'#dee2e6'	Component borders (--bs-border-color).
FontFamily	string	(system stack)	Body font stack (--bs-font-sans-serif).
BorderRadius	string	'0.375rem'	Corner radius (--bs-border-radius).
Mode	TsgcHTMLTheme-Mode	tmLight	tmLight, tmDark or tmSystem.

Methods

- **GetHTML: string** — returns a <style> block (and a <script> block for tmSystem).
- **GetCSS: string** — returns the CSS variable overrides without the <style> tags.
- **GetScript: string** — returns the OS-preference sync script used by tmSystem.
- **SetLightDefaults / SetDarkDefaults / SetSystemDefaults** — load a complete built-in palette and set Mode.

Output

Assign the builder to TsgcHTMLTemplate_Bootstrap.ThemeBuilder to have its CSS injected automatically, or call GetHTML and append the result to CustomHead yourself.

Reference

- Properties — 13 properties available on the component.
- Methods — 6 public methods.
- Events — this component has no published events.

TsgcHTMX_Fragment

TsgcHTMX_Fragment builds an out-of-band htmx swap string for a specific DOM element. Set TargetID and Content, read Fragment, and pass it to TsgcHTMX_Engine_Server.BroadcastFragment or return it as the HTTP response body.

Introduction

TsgcHTMX_Fragment assembles the out-of-band (OOB) htmx swap payload that htmx uses to update a specific element on the page without a full reload. Set TargetID to the id of the element to replace, choose a SwapMethod (innerHTML, outerHTML, beforebegin, etc.) and assign the new HTML to Content. The Fragment property then returns the complete OOB swap string.

Pass the Fragment to TsgcHTMX_Engine_Server.BroadcastFragment for server-push updates, or return it directly as the HTTP response body for htmx-triggered requests. See also TsgcHTMX_Router.

Quick Start

1. Create a TsgcHTMX_Fragment.

2. Set TargetID, SwapMethod and Content.

3. Read Fragment and pass it to BroadcastFragment or write it as the response.

```
TsgcHTMX_Fragment *oFrag = new TsgcHTMX_Fragment(NULL);
try
{
  oFrag->TargetID   = "kpi-revenue";
  oFrag->SwapMethod = "innerHTML";
  oFrag->Content    = FormatFloat("#,##0", FRevenue);
  oHTMX->BroadcastFragment(oFrag->Fragment);
}
__finally { delete oFrag; }
```

Fragment Settings

What it does: Assembles the OOB htmx swap payload for a named DOM element.

Property	Type	Default	Description
TargetID	string	''	id of the element GetOOBFragment updates (the hx-swap-oob target). Filtered to the characters that are valid in an element id.
SwapMode	TsgcHTMXSwapMode	hxInnerHTML	htmx swap strategy: hxInnerHTML, hxOuterHTML, hxBeforeBegin, hxAfterBegin, hxBeforeEnd, hxAfterEnd, hxDelete, hxNone or hxMorph.

SwapOOB	Boolean	True	Writes the hx-swap-oob attribute in the HTML property. GetOOBFragment always writes it.
Content	string	''	New HTML to inject into the target element.
Tag	string	'div'	Element the fragment is wrapped in. A value that is not an element name falls back to div.
CSSID	string	''	id written by the HTML property, and the fallback of GetOOBFragment when TargetID is empty. Filtered in the same way.
CSSClass	string	''	Classes of the wrapper. Attribute-encoded when it is written.

Output

GetOOBFragment returns a string of the form `<div id="{TargetID}" hx-swap-oob="{swap}">{Content}</div>`, which is what PushFragment and BroadcastFragment send. The inherited **HTML** property returns the same element with the id of *CSSID*, and with the hx-swap-oob attribute only when *SwapOOB* is True. The class function **Build**(aID, aContent, aSwapMode) returns the out-of-band form in one call.

hxMorph emits `hx-swap-oob="morph"`: the target element is updated in place by the idiomorph extension, which the page loads when its template has a DefaultSwap that starts with `morph`, so the focus and the value being typed survive the update. An element marked with `data-sgc-preserve` is left untouched while it has the focus. See Navigation and Live Updates.

Reference

- Properties — 7 properties available on the component.
- Methods — 2 public methods: GetOOBFragment and the class function Build.
- Events — this component has no published events.

TsgcHTMX_Router

TsgcHTMX_Router maps URL patterns and HTTP methods to Delphi handler procedures, eliminating if/else chains in a request dispatcher.

Introduction

TsgcHTMX_Router provides a lightweight URL router for htmx-driven Delphi HTTP servers. Add a route with **Routes.Add** and set its **Path** (a literal path or a template with `{name}` segments), optionally its **Methods** (GET, POST, PUT, PATCH, DELETE) and its access rules (**RequireLogin**, **RequireRoles**), and its **OnRoute** handler, which returns the HTML fragment. Assign the router to the Router property of TsgcHTMX_Engine_Server and the engine runs the matching route for HTTP requests and htmx WebSocket messages.

If no route matches, or the handler returns an empty string, the request is not handled and the host can serve a 404. See Routing for the matching rules. See also TsgcHTMX_Fragment.

Quick Start

1. Create a TsgcHTMX_Router.

2. Call Routes.Add for each URL pattern and set Path, Methods and OnRoute.

3. Assign the router to the Router property of the engine.

```
oRouter = new TsgcHTMX_Router(this);

TsgcHTMX_Route *oRoute = oRouter->Routes->Add();
oRoute->Path = "/customers";
oRoute->OnRoute = HandleCustomers;

oRoute = oRouter->Routes->Add();
oRoute->Path = "/customer/{id}";
oRoute->Methods = TsgcHTMXRouteMethods() << hrmPost;
oRoute->OnRoute = HandleCustomerSave;

oHTMX->Router = oRouter; // TsgcHTMX_Engine_Server
```

Router

What it does: Holds the ordered list of registered routes.

Property	Type	Default	Description
Routes	TsgcHTMX_Routes	(empty)	The ordered list of registered routes.

Each route added with Routes.Add is a TsgcHTMX_Route:

Property	Type	Default	Description
Path	string	''	Literal path, or template with whole {name} segments whose values arrive in aRequest.Params.
Methods	TsgcHTMXRouteMethods	[]	hrmGet, hrmPost, hrmPut, hrmPatch, hrmDelete. Empty answers GET on the Indy and http.sys engines, every method on WebBroker and any WebSocket message.
RequireLogin	Boolean	False	Only a signed-in user (a session of the engine Auth) reaches the route.
RequireRoles	string	''	Comma separated roles; the session needs one of them. Implies RequireLogin.
OnRoute	TsgcHTMXRouteEvent	nil	Handler that sets aResponse to the HTML to return.

Methods

- **HandleRequest(aPath, aMethod, aGuid: string; aParams: TStringList): string**: finds the route of a decoded path and runs it. A route with an empty Methods set matches any method here.
- **HandleRequest(aRequest: TsgcHTMXDispatchRequest; aGuid: string; aUnfiltered: Boolean): string**: the same for a transport-neutral request; aUnfiltered decides whether routes with an empty Methods set match.
- **FindRoute(aRequest: TsgcHTMXDispatchRequest; aUnfiltered: Boolean): TsgcHTMX_Route**: returns the matching route, or nil.
- **HandleRoute(aRoute, aRequest, aGuid, aSession): string**: runs a route found with FindRoute, passing an optional session.

Output

Literal routes are matched first and template routes next, each group in registration order. The first route that matches the path and accepts the method runs, and HandleRequest returns the aResponse of its handler, or an empty string when no route applies. The router applies no access rule: RequireLogin, RequireRoles and the CSRF check are the job of the engine (see Sessions, CSRF and Authentication).

Reference

- Properties — 1 property available on the component.
- Methods — 3 public methods.
- Events — this component has no published events.

Demos | Server Chat

This demo shows how to build a Server Chat using TsgcWebSocketHTTPServer and WebSockets as communication protocol.
Every time a new peer sends a message, the server reads the message and broadcast the message to all connected clients.

Start Server

Before start the server, you must configure it to set the listening port, if use a secure connection or not...

- First I create a new instance of TsgcWebSocketHTTPServer.
- If Server will use secure connections, it needs a PEM certificate, just set where is located this certificate and the listening port for SSL You can configure the TLS version and the OpenSSL API (if needed)

```
// ... ssl
switch (cboOpenSSLAPI->ItemIndex)
{
        case 0:
                server->SSLOptions->OpenSSL_Options.APIVersion = TwsOpenSSLAPI->oslAPI_1_0;
                break;
        case 1:
                server->SSLOptions->OpenSSL_Options->APIVersion = TwsOpenSSLAPI->oslAPI_1_1;
                break;
}
switch (cboTLSVersion->ItemIndex)
{
        case 0:
                server->SSLOptions->Version = TwsTLSVersions->tlsUndefined;
                break;
        case 1:
                server->SSLOptions->Version = TwsTLSVersions->tls1_0;
                break;
        case 2:
                server->SSLOptions->Version = TwsTLSVersions->tls1_1;
                break;
        case 3:
                server->SSLOptions->Version = TwsTLSVersions->tls1_2;
                break;
        case 4:
                server->SSLOptions->Version = TwsTLSVersions->tls1_3;
                break;
        default:
                break;
}
```

- By default, if you start the server, it will listening on ALL IPs of listening port, so it's recommended use the binding property to only listen on 1 specific IP.

```
With WSServer->Bindings->Add do
{
  Port = StrToInt(txtDefaultPort->Text);
  IP = txtHost->Text;
}
```

- Once configured all options, call Server.Active = true to start the server.

Events Configuration

- Use **OnConnect** and **OnDisconnect** events to know when a client connects to server.
- **Messages** sent from **client to server** are received **OnMessage** event, so use this event handler to broadcast the message received to all clients

```
private void OnMessageEvent(TsgcWSConnection *Connection, string Text)
{
  server->Broadcast(Text);
}
```

Dispatch HTTP Requests

WebSocket HTTP Server allows you to handle WebSocket and HTTP Protocols on the same listening port, so a web-browser can request a web page to access your server. **OnCommandGet** is the event used to read the HTTP Request and send the HTTP Responses.

Use ARequestInfo parameter to read the HTTP Request and AResponseInfo to write the HTTP Response.

Basically, use the ARequestInfo.Document to read which document is requesting the client and send a response using the following properties: ResponseNo, ContentType and ContentText.

Example: a client request document '/jquery.js'

```
private void OnCommandGetEvent(TsgcWSConnection *Connection, TsgcWSHTTPRequestInfo *RequestInfo, ref TsgcWSHTTPRe
{
        if (RequestInfo->Document == "/jquery.js")
        {
                ResponseInfo->ContentType = "text/javascript";
                ResponseInfo->ContentText = pageJQuery.Content;
                ResponseInfo->ResponseNo = 200;
        }
}
```

Client Chat

This demo shows how to build a client chat, using TsgcWebSocketClient, which connects to a WebSocket Server, sends a message and this message is received by all connected clients.

Connect to Server

- First create a new instance of TsgcWebSocketClient.
- Then configure the server **Host** and **Port**.
- If client uses a secure connection, configure the **TLSOptions** property of the component.

```
if (chkTLS->Checked)
{
  WSClient->Port = StrToInt(txtSSLPort->Text)
}
else
{
  WSClient->Port = StrToInt(txtDefaultPort->Text);
}
WSClient->Host = txtHost->Text;
case cboOpenSSLAPI->ItemIndex of
{
  0: WSClient->TLSOptions->OpenSSL_Options->APIVersion = oslAPI_1_0;
  1: WSClient->TLSOptions->OpenSSL_Options->APIVersion = oslAPI_1_1;
};
case cboTLSVersion->ItemIndex of
{
  0: WSClient->TLSOptions->Version = tlsUndefined;
  1: WSClient->TLSOptions->Version = tls1_0;
  2: WSClient->TLSOptions->Version = tls1_1;
  3: WSClient->TLSOptions->Version = tls1_2;
  4: WSClient->TLSOptions->Version = tls1_3;
}
WSClient->TLS = chkTLS->Checked;
```

- Once all options can be configured, set Client.Active = true to connect to server.

Send Message To Server

- To send a message to server, use **WriteData** method, send any Text message and server will send as a response the same message.

```
WSClient->WriteData(txtName->Text + ": " + txtMessage->Text);
```

Receive Messages from Server

- Every time a new Text message is received by client, **OnMessage** event is fired.

```
private void OnMessageEvent(TsgcWSConnection *Connection, string Text)
{
        DoLog(Text);
}
```

Demos | Client

This demo shows how to build a websocket client, using TsgcWebSocketClient.

Connect to Server

- First create a new instance of TsgcWebSocketClient.
- Then configure the server **Host** and **Port**.
- By default the client will connect using **WebSocket protocol**. But you can configure the client to connect using **plain TCP protocol**. Just set **Specifications.RFC6455 = false**, and the client will use plain TCP protocol instead of WebSocket protocol. You can read more about TCP Connections.

```
WSClient->Host = txtHost->Text;
WSClient->Port = StrToInt(txtPort->Text);
WSClient->Options->Parameters = txtParameters->Text;
WSClient->TLS = chkTLS->Checked;
WSClient->Specifications->RFC6455 = chkOverWebSocket->Checked;
WSClient->Proxy->Enabled = chkProxy->Checked;
WSClient->Proxy->Username = txtProxyUsername->Text;
WSClient->Proxy->Password = txtProxyPassword->Text;
WSClient->Proxy->Host = txtProxyHost->Text;
if txtProxyPort->Text  "" then
WSClient->Proxy->Port = StrToInt(txtProxyPort->Text);
WSClient->Extensions->PerMessage_Deflate->Enabled := chkCompressed->Checked;
// ... active
WSClient->Active = True;
```

Client Events

Use the following events to control the client flow: when connects, disconnects, receives a message, an error is detected...

```
private void OnExceptionEvent(TsgcWSConnection *Connection, Exception E)
{
        DoLog("#exception: " + E->Message);
}
private void OnConnectEvent(TsgcWSConnection *Connection)
{
        DoLog("#connected: " + Connection->IP);
}
private void OnMessageEvent(TsgcWSConnection *Connection, string Text)
{
        DoLog(Text);
}
private void OnDisconnectEvent(TsgcWSConnection *Connection, int CloseCode)
{
        DoLog("Disconnected (" + IntToStr(CloseCode) + "): " + Connection->IP);
}
private void OnErrorEvent(TsgcWSConnection *Connection, string Error)
{
        DoLog("#error: " + Connection->IP + " - " + Error);
}
```

Demos | Client MQTT

This demo shows how connect to a MQTT broker server. Requires a TsgcWebSocketClient to handle WebSocket / TCP protocols.

Configuration

- First create a new TsgcWebSocketClient instance, check the Client Demo.
- Then, create a new instance of TsgcWSPClient_MQTT.
- After that, you must assign the MQTT Protocol to WebSocket client and configure the connection options in WebSocket client.

```
if (mqtt == null)
{
        mqtt = new TsgcWSPClient_MQTT();
        mqtt->OnMQTTBeforeConnect += OnMQTTBeforeConnectEvent;
        mqtt->OnMQTTConnect += OnMQTTConnectEvent;
        mqtt->OnMQTTDisconnect += OnMQTTDisconnectEvent;
        mqtt->OnMQTTSubscribe += OnMQTTSubscribeEvent;
        mqtt->OnMQTTUnSubscribe += OnMQTTUnSubscribeEvent;
        mqtt->OnMQTTPing += OnMQTTPingEvent;
        mqtt->OnMQTTPubAck += OnMQTTPubAckEvent;
        mqtt->OnMQTTPubComp += OnMQTTPubCompEvent;
        mqtt->OnMQTTPublish += OnMQTTPublishEvent;
        mqtt->OnMQTTPubRec += OnMQTTPubRecEvent;
        mqtt->Client = client;
}
mqtt->Client = client;
txtParameters->Text = "/";
chkTLS->Checked = false;
mqtt->Authentication->Enabled = false;
mqtt->Authentication->UserName = "";
mqtt->Authentication->Password = "";
mqtt->MQTTVersion = TwsMQTTVersion->mqtt311;
mqtt->HeartBeat->Interval = 5;
mqtt->HeartBeat->Enabled = true;
switch (Index)
{
        case 0: // esegece.com
                txtHost->Text = "www.esegece.com";
                txtPort->Text = "15675";
                txtParameters->Text = "/ws";
                mqtt->Authentication->Enabled = true;
                mqtt->Authentication->UserName = "sgc";
                mqtt->Authentication->Password = "sgc";
                chkOverWebSocket->Checked = true;
                break;
        case 1: // test.mosquitto.org
                txtHost->Text = "test.mosquitto.org";
                txtPort->Text = "1883";
                chkTLS->Checked = false;
                chkOverWebSocket->Checked = false;
                break;
        case 2: // mqtt.fluux.io
                txtHost->Text = "mqtt.fluux.io";
                txtPort->Text = "1883";
                chkTLS->Checked = false;
                chkOverWebSocket->Checked = false;
                mqtt->MQTTVersion = TwsMQTTVersion->mqtt5;
                break;
        case 3: // broker.hivemq.com
                txtHost->Text = "broker.mqttdashboard.com";
                txtPort->Text = "8000";
                txtParameters->Text = "/mqtt";
                chkTLS->Checked = false;
                chkOverWebSocket->Checked = true;
                mqtt->MQTTVersion = TwsMQTTVersion->mqtt5;
                break;
}
```

MQTT Events

The connection flow is controlled by MQTT Client component, so you must handle the MQTT events to know when it's connected to broker, when a new message is published, when is disconnected...

```
private void OnMQTTConnectEvent(TsgcWSConnection *Connection, bool Session, int ReasonCode, string ReasonName, Ts
{
        DoLog("#MQTT Connect");
        chkTLS->Enabled = false;
        chkCompressed->Enabled = false;
        if (FMQTTSubscribeTopic != "")
        {
                mqtt->Subscribe(FMQTTSubscribeTopic);
                FMQTTSubscribeTopic = "";
        }
}



private void OnMQTTPublishEvent(TsgcWSConnection *Connection, string Topic, string Text, TsgcWSMQTTPublishPropert
{
        DoLog(Topic + ": " + Text);
}



private void OnMQTTSubscribeEvent(TsgcWSConnection *Connection, int PacketIdentifier, TsgcWSSUBACKS *Codes, TsgcW
{
        DoLog("#Subscribe: " + IntToStr(PacketIdentifier));
}



private void OnMQTTDisconnectEvent(TsgcWSConnection *Connection, int ReasonCode, string ReasonName, TsgcWSMQTTDIS
{
        DoLog("#disconnected");
        chkTLS->Enabled = true;
        chkCompressed->Enabled = true;
}
```

Demos | Client SocketIO

This demo shows how connect to a Socket.IO Server. Requires a TsgcWebSocketClient to handle WebSocket / TCP protocols.

Configuration

- First create a new TsgcWebSocketClient instance, check the Client Demo.
- Then, create a new instance of TsgcWSAPI_SocketIO.
- After that, you must assign the Socket.IO API to WebSocket client and configure the connection options in WebSocket client.

```
if (socketio == null)
{
        socketio = new TsgcWSAPI_SocketIO();
        socketio->Client = client;
}
socketio->Client = client;
txtParameters->Text = "/";
chkTLS->Checked = true;
chkOverWebSocket->Checked = true;
```

Send Messages

Socket.IO uses **TsgcWebSocketClient** to send messages to server, so just call **WriteData** and pass as a parameter the JSON message to socket.io server

```
client->WriteData("42[\"new message\", \"" +  txtSocketIOMessage.Text + "\"]");
```

Receive Messages

The messages received as the flow of connection is handled by TsgcWebSocketClient, so use this component to read the messages sent from server and to know if connection is active or not.

```
private void OnMessageEvent(TsgcWSConnection *Connection, string Text)
{
  DoLog(Text);
}
```

Demos | Server Monitor

This demo shows how to update 3 HTML Monitors using WebSocket as protocol. Server has an internal timer that updates randomly the values of the gauges and updates the value using a websocket message. This message is read by javascript client and updates the value of the Gauge.

Configuration

- First create a new TsgcWebSocketServer instance, check the Server Chat Demo.
- Then Create a new Timer and every 500 milliseconds update the values of: memory, network or cpu. Send the update to all clients connected.
- In Javascript client, read the message sent by server and update the value of the gauge.

```
<pre><code>
<!DOCTYPE html>
<html>
<head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Server Monitor Demo</title>
        <script src="http://127.0.0.1:5413/sgcWebSockets.js"></script>
        <script src="http://127.0.0.1:5413/esegece.com.js"></script>
        <link rel="stylesheet" href="http://code.jquery.com/mobile/1.1.0/jquery.mobile-1.1.0.min.css" />
        <script src="http://code.jquery.com/jquery-1.6.4.min.js"></script>
        <script src="http://code.jquery.com/mobile/1.1.0/jquery.mobile-1.1.0.min.js"></script>
        <script type='text/javascript' src='https://www.google.com/jsapi'></script>
        <style>
        #status {
                padding: 5px;
                color: #fff;
                background: #ccc;
        }
        #status.fail {
          background: #c00;
        }
        #status.offline {
          background: #c00;
        }
        #status.online {
          background: #0c0;
        }
        </style>
        <script type='text/javascript'>
          var vMemory;
          var vCpu;
          var vNetwork;
          var chart;
          var data;
          var options;
          var ws;

          vMemory=30;
          vCpu=55;
          vNetwork=68;
      google.load('visualization', '1', {packages:['gauge']});
          google.setOnLoadCallback(drawChart);
      function drawChart() {
        data = google.visualization.arrayToDataTable([
          ['Label', 'Value'],
          ['Memory', vMemory],
          ['CPU', vCpu],
          ['Network', vNetwork]
        ]);
        options = {
          width: 400, height: 120,
          redFrom: 90, redTo: 100,
          yellowFrom:75, yellowTo: 90,
          minorTicks: 5
        };
        chart = new google.visualization.Gauge(document.getElementById('chart_div'));
        chart.draw(data, options);
        }

        function updateChart() {
```

```
                data = google.visualization.arrayToDataTable([
                        ['Label', 'Value'],
                        ['Memory', vMemory],
                        ['CPU', vCpu],
                        ['Network', vNetwork]
                ]);
                chart.draw(data, options);
        }

        function subscribe(Channel)
        {
          if (document.getElementById(Channel).checked) {
                ws.subscribe(Channel);
          } else {
                ws.unsubscribe(Channel);
          }
        }

        function wsmonitor()
        {
          if ("WebSocket" in window)
          {
                  ws = new sgcws("ws://127.0.0.1:5413");
                  ws.on('open', function(evt){
                                document.getElementById('status').innerHTML = "Socket Open";
                                document.getElementById('status').className = "online";
                                ws.subscribe("memory");
                                ws.subscribe("cpu");
                                ws.subscribe("network");
                        }
                  );
                  ws.on('close', function(evt){
                                document.getElementById('status').innerHTML = "Socket Closed";
                                document.getElementById('status').className = "offline";
                        }
                  );
                  ws.on('sgcevent', function(evt){
                        if (evt.channel == "memory") {
                          vMemory = parseInt(evt.message);
                        } else if (evt.channel == "cpu") {
                          vCpu = parseInt(evt.message);
                        } else if (evt.channel == "network") {
                          vNetwork = parseInt(evt.message);
                        }
                        updateChart();
                        }
                  );
                  ws.on('error', function(evt){
                                document.getElementById('status').innerHTML = "Socket Error";
                                document.getElementById('status').className = "fail";
                        }
                  );
          }
        }
</script>
</head>
<body>
<div data-role="page" id="wsdemo_monitor">
        <div data-role="header" data-theme="b">
                <h1>Server Monitor</h1>
                <a href="#home" data-icon="home" data-iconpos="notext" data-direction="reverse" class="ui-btn-lef
        </div><!-- /header -->
        <div data-role="content">
                <h2>Press Start to Get Monitor Data</h2>
                <p id="status" classname="success"></p>
                <h4>Select which data you want to receive: Memory - CPU - Network</h4>
                <a href="javascript:wsmonitor()" data-role="button" data-inline="true">Start</a>
                <div id='chart_div'></div>
                <div data-role="fieldcontain">
                    <fieldset data-role="controlgroup" data-type="horizontal">
                            <input type="checkbox" name="memory" id="memory" class="custom" checked="True" onclic
                                <label for="memory">Memory</label>
                                <input type="checkbox" name="cpu" id="cpu" class="custom" checked="True" onclick=
                                <label for="cpu">CPU</label>
                                <input type="checkbox" name="network" id="network" class="custom" checked="True"
                                <label for="network">Network</label>
                    </fieldset>
                </div>
        </div><!-- /content -->
        <div data-role="footer" class="footer-docs" data-theme="c">
                        <p>&copy; 2020 eSeGeCe.com</p>
        </div>
</div><!-- /page -->
</body>
</html>
</code></pre>
```

Demos | Server Snapshots

This demo shows how to send images from server to client and how all clients receive the same image using broadcast method of server component.

Configuration

- First create a new TsgcWebSocketHTTPServer instance, check the Server Chat Demo.
- Enable compression to send less bytes when message is transmitted to clients

  Server.Extensions.PerMessage_Deflate.Enabled = true

- Then every 5 seconds the server broadcast an image stream to all connected clients

```
void DoBroadcastStream()
{
  TBitmap *oBitmap = new TBitmap();
  try
  {
    // load bitmmap
    if (WSServer->Active)
    {
      TMemoryStream *oStream = new TMemoryStream();
      Try
          {
        oBitmap->SaveToStream(oStream);
        oStream->Seek(0, soFromBeginning);
        WSServer->Broadcast(oStream);
          }
      __finally
          {
        FreeAndNil(oStream);
      }
    }
  }
  __finally
  {
    oBitmap->FreeImage;
    FreeAndNil(oBitmap);
  }
}
```

Demos | Client Snapshots

This demo shows how read binary websocket messages, using TsgcWebSocketClient, which connects to a Web-Socket Server, and receives a stream which is an image that is shown to user.

Connect to Server

- First create a new instance of TsgcWebSocketClient.
- Then configure the server **Host** and **Port**.
- Enable compression to receive less bytes when message is transmitted from server.

```
Client.Extensions.PerMessage_Deflate.Enabled = true
```

- The image sent by server arrives as a stream, so use **OnBinary** event to read images.

```
void OnClientBinary(TsgcWSConnection *Connection, TMemoryStream *Data)
{
  TBitmap *oBitmap = new TBitmap();
  try
  {
    oBitmap->LoadFromStream(Data);
    Image1->Picture->Assign(oBitmap);
    memoLog->Lines->Add(
      "#image uncompressed size: " + IntToStr(Data->Size) +
      ". Total received: " + IntToStr(Connection->RecBytes));
  }
  __finally
  {
    FreeAndNil(oBitmap);
  }
}
```

Demos | Upload File

This demo shows how to upload a file from web browser to a server using websocket protocol.

Configuration

- First create a new TsgcWebSocketServer instance, check the Server Chat Demo.
- The file will arrive to server as a binary stream, so you must handle **OnBinary** event to read the file.

```
void OnServerBinary(TsgcWSConnection *Connection, const TMemoryStream *Data)
{
  if (FFileName == "")
    FFileName = FormatDateTime("yyyymmddhhnnsszz", Now) + ".dat";
  TFileStream *oFile = new TFileStream(FFileName, fmCreate);
  try
  {
    oFile->CopyFrom(Data, Data->Size);
  }
  __finally
  {
    delete oFile;
  }
  memoLog->Lines->Add("Received File: " + FFileName);
}
```

- If you want to know the name of the file, you can send a text message before the file is sent with the name of the file

```
void OnServerMessage(TsgcWSConnection *Connection, const String Text)
{
  if (LeftStr(Text, Length(CS_UPLOAD_FILE)) == CS_UPLOAD_FILE)
    FFileName = MidStr(Text, Length(CS_UPLOAD_FILE) + 1, Length(Text));
  else
    memoLog->Lines->Add("Message Received (" + Connection->Guid + "): " + Text);
}
```

The **javascript** code to send a file using **websockets** is shown below:

```
<script type='text/javascript'>
var ws;
function DoOpen()
{
  if ("WebSocket" in window)
  {
          ws = new sgcWebSocket("ws://127.0.0.1:5418");
          ws.on('open', function(evt){
                        ws.binaryType = "arraybuffer";
                        document.getElementById('status').innerHTML = "Socket Open";
                        document.getElementById('status').className = "online";
                }
          );
          ws.on('close', function(evt){
                        document.getElementById('status').innerHTML = "Socket Closed";
                        document.getElementById('status').className = "offline";
                }
          );
          ws.on('error', function(evt){
                        document.getElementById('status').innerHTML = "Socket Error";
                        document.getElementById('status').className = "fail";
                }
          );
  }
}
function DoClose()
{
        ws.close();
}
function DoUploadFile() {
```

```
        var file = document.getElementById('filename').files[0];
        var reader = new FileReader();
        var rawData = new ArrayBuffer();

        reader.loadend = function() {

        }
        reader.onload = function(e) {
                ws.send("uploadfile:" + file.name);
                rawData = e.target.result;
                ws.send(rawData);
                        document.getElementById('status').innerHTML = "File Uploaded";
                        document.getElementById('status').className = "online";
                }
                reader.readAsArrayBuffer(file);
        }
</script>
```

Demos | Server Authentication

This demo shows how to use Server Authentication, if you want to know more about the different types of authentication, read the following article about Authentication.

Authentication

- First create a new instance of TsgcWebSocketServer. Enable Authentication property, server.Authentication.Enabled = true;
- Then, check in **OnAuthentication** event handler if the username and password are correct. If they are correct, set the Authenticated property to true, otherwise set to false.

```
private void OnAuthenticationEvent(TsgcWSConnection *Connection, string User, string Password, ref bool Authentic
{
        if ((User == "user") && (Password == "1234"))
        {
                Authenticated = true;
        }
}
```

Demos | KendoUI_Grid

This demo shows how the KendoUI Grid works using WebSockets as protocol and a Web Browser as a client. Basically is a javascript grid that is updated when any of the clients makes any change, these changes are updated using websocket protocol to all connected clients, so all clients can see in real-time the same data, including all changes made by clients.

Configuration

- First create a new TsgcWebSocketHTTPServer instance, check the Server Chat Demo.
- Then you must handle OnCommandGet to send the required files requested by web browser clients.

```
void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo *AResponseInfo)
{
  if (ARequestInfo->Document == "/jquery.mobile.css")
  {
    AResponseInfo->ContentText = pageJQueryMobileCSS->Content;
    AResponseInfo->ContentType = "text/css";
    AResponseInfo->ResponseNo = 200;
  }
  else if (ARequestInfo->Document == "/jquery.js")
  {
    AResponseInfo->ContentText = pageJQuery->Content;
    AResponseInfo->ContentType = "text/javascript";
    AResponseInfo->ResponseNo = 200;
  }
  else if (ARequestInfo->Document == "/jquery.mobile.js")
  {
    AResponseInfo->ContentText = pageJQueryMobile->Content;
    AResponseInfo->ContentType = "text/javascript";
    AResponseInfo->ResponseNo = 200;
  }
  else
  {
    if (AContext->Connection->Socket->Binding->Port == WSServer->SSLOptions->Port)
      FRequestSSL = true;
    else
      FRequestSSL = false;
    AResponseInfo->ContentText = pageKendoUI_Grid->Content;
    AResponseInfo->ContentType = "text/html";
    AResponseInfo->ResponseNo = 200;
  }
}
```

WebSockets Updates

When a client updates a grid record, this change is transmitted to all connected clients using websocket protocol. Use OnMessage event to get notified about grid changes. The messages are in JSON format so you only must read the JSON text, decode it and send a response to the other peer.

```
void OnServerMessage(TsgcWSConnection *Connection, const String Text)
{
  TsgcJSON *oJSON = new TsgcJSON();
  try
  {
    oJSON->Read(Text);
    // ... read
    if (oJSON->Node["type"]->Value == "read")
    {
      IsgcObjectJSON *oArray = oJSON->AddArray("data");
      for (int i = 0; i <20; i++)


      {
        TsgcJSON *oObj = oArray->JSONObject->AddObject(IntToStr(i))->JSONObject;
        oObj->AddPair("ContactID", i);
          oObj->AddPair("ContactName", ContactName[i]);
```

```
          oObj->AddPair("ContactTitle", ContactTitle[i]);
          oObj->AddPair("CompanyName", CompanyName[i]);
          oObj->AddPair("Country", Country[i]);
      }
      Connection->WriteData(oJSON->Text);
    }
    // ... update
    else if (oJSON->Node["type"]->Value == "update")
    {
      WSServer->Broadcast(StringReplace(Text, "\"type\":\"update\"", "\"type\":\"push-update\"", []), "", "",
        Connection->Guid);
      Connection->WriteData(Text);
    }
    // ... destroy
    else if (oJSON->Node["type"]->Value == "destroy")
    {
      WSServer->Broadcast(StringReplace(Text, "\"type\":\"destroy\"", "\"type\":\"push-destroy\"", []), "", "",
        Connection->Guid);
      Connection->WriteData(Text);
    }
    // ... create
    else if (oJSON->Node["type"]->Value == "create")
    {
      String vText = StringReplace(Text, "null", FormatDateTime("yyyymmddhhnnsszzz", Now), []);
      WSServer->Broadcast(StringReplace(vText, "\"type\":\"create\"", "\"type\":\"push-create\"", []), "", "",
        Connection->Guid);
      Connection->WriteData(vText);
    }
  }
  __finally
  {
    oJSON->Free();
  }
}
```

Demos | ServerSentEvents

This demo shows how Server-Sent Events work in WebSocket Server. sgcWebSockets allows that the server can handle more than one protocol on the same listening port.

You can read more about Server Sent Events.

This demo shows how the Server will send every second the time to all connected clients using Server Sent Events.

Once the server is started, **broadcasts** to all connected clients a message with the **Server Time**, so every time the client receives this message, it shows to user.

```
private void Timer1Timer(TObject *sender)
{
  server->Broadcast("data: " + "Server Time: " + FormatDateTime("hh:nn:ss", Now));
}
```

The **javascript code** to handle the websocket connection is shown below:

```
socket = new sgcWebSocket('sse', '', 'sse');
socket.on('open', function(evt){
  document.getElementById('status').innerHTML = "Socket Open";
  document.getElementById('status').className = "online";
  }
);
socket.on('close', function(evt){
  document.getElementById('status').innerHTML = "Socket Closed";
  document.getElementById('status').className = "offline";
  }
);
socket.on('message', function(evt){
  document.getElementById('log').innerHTML = evt.message;
  }
);
socket.on('error', function(evt){
  document.getElementById('status').innerHTML = "Socket Error";
  document.getElementById('status').className = "fail";
  }
);
```

Demos | Server WebRTC

This demo shows how to build a Video Conference Server using TsgcWebSocketHTTPServer and WebRTC as javascript library.

The demo uses WebSocket protocol to signal WebRTC and uses public STUN/TURN servers, for production sites, you need to use your own STUN/TURN servers. Registered users can download Coturn for windows, which is already compiled and with all the required libraries to run in your servers.

The client must be a web browser with support for WebRTC connections.

Configuration

- First create a new TsgcWebSocketHTTPServer instance, check the Server Chat Demo.
- Then, create a new instance of TsgcWSPServer_WebRTC.
- After that, you must assign the WebRTC Protocol to WebSocket Server and configure the server host and port.

```
WSServer->Port = StrToInt(txtDefaultPort->Text);
// ... bindings
With WSServer->Bindings->Add do
{
  Port = StrToInt(txtDefaultPort->Text);
  IP = txtHost>Text;
}
// ... active
WSServer->Active = true;
```

The demo requires an index HTML page which is used to dispatch the WebRTC front page, this page is provided with the demo.

Run in WebBrowser

Once configured the server, start it and select one of the web-browsers available. It will open a new Web-Browser session asking to start a new session. If successful you will see your video and if you open the same url in another web-browser, you will see both peers connected.

The demo runs by default without SSL, this is only valid for localhost connections. For production sites, use SSL connections. Check Server Chat Demo to configure SSL in server side.

Demos | Telegram Client

This demo shows how to connect to Telegram, receive all contacts, send Text messages, send Images... and much more

Configuration

- First create a new instance of TsgcTDLib_Telegram.
- Then, before you attempt to connect to telegram, you must pass some parameters to client component like API Hash, API Id... Once you must set all required parameters, set property Active = true to start a connection.

```
telegram->Telegram->API->ApiHash = txtApiHash->Text;
telegram->Telegram->API->ApiId = txtApiId->Text;
telegram->Telegram->PhoneNumber = "";
telegram->Telegram->BotToken = "";
if (chkLoginBot->Checked)
{
  telegram->Telegram->BotToken = txtBotToken->Text;
}
else
{
  telegram->Telegram->PhoneNumber = txtPhoneNumber->Text;
}


telegram->Active = true;
```

- When client tries to connect to Telegram, usually a code is required, so you must handle **OnTelegramAuthenticationCode** and return the Code parameter with the value provided by your Telegram account.

```
private void OnAuthenticationCodeEvent(TObject *Sender, ref string Code)
{
  Code = InputBox("Telegram", "Introduce Telegram Code");
}
```

Send Telegram Messages

To send a telegram message (text, files, images...) always requires first set the ChaId where you want to send the message and then the parameter that can be a text message, a filename...

```
// send text message
sgcTelegram->SendTextMessage("456413", "Hello From sgcWebSockets!!!");


// send file message
sgcTelegram->SendDocumentMessage("383784", "c:\yourfile.txt");
```

Receive Telegram Messages

Messages received by Telegram client, are handled on specific event Handlers. There is an event when a next Text Message is received, when a new Document is received, photo...

```
private void OnMessageTextEvent(TObject *Sender, TsgcTelegramMessageText *MessageText)
{
  DoLogMessage(MessageText->ChatId, MessageText->SenderUserId->ToString(), MessageText->Text);
```

```
}


private void OnMessageDocumentEvent(TsgcTDLib_Telegram *Sender, TsgcTelegramMessageDocument *MessageDocument)
{
  DoLogMessage(MessageDocument->ChatId, MessageDocument->SenderUserId->ToString(), MessageDocument->FileName);
}
```

Coturn

Coturn

From sgcWebSockets 4.5.2 ENTERPRISE Edition, you can build your own STUN/TURN server using Delphi/CBuilder.

It's a free open source implementation of TURN and STUN Servers.
The TURN Server is a VoIP media traffic NAT traversal server and gateway. It can be used as a general-purpose network traffic TURN server and gateway, too.

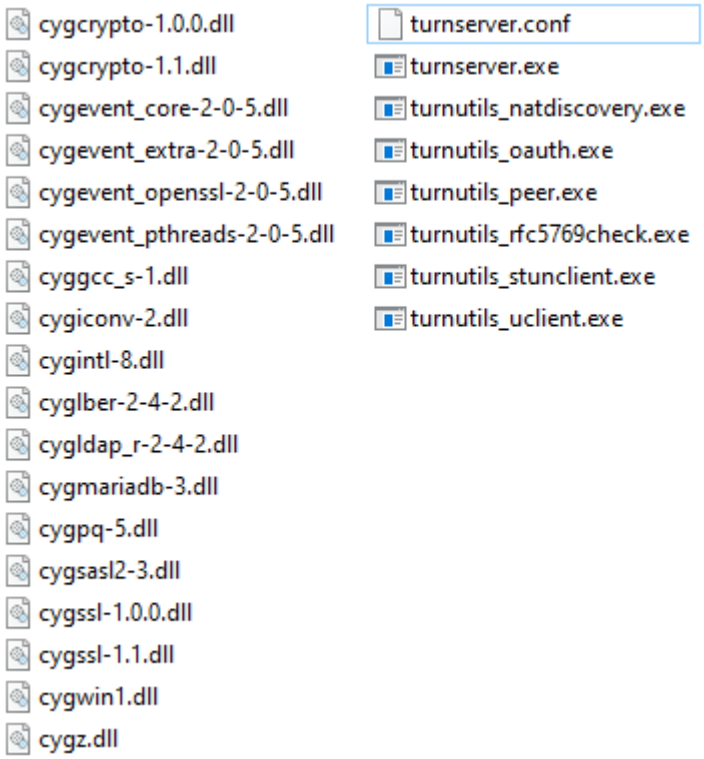
The supported project target platforms are:

- **Linux**
- **Mac OS X**
- **Windows** (Cygwin): compiled binaries are available for registered users.

Windows Configuration

First you must download compiled binaries from your account, there are 2 available versions: win32 and win64. Select the desired platform and uncompress binaries in a folder. The following files will be created:

1. Some cygwin libraries required to run application, you must deploy these libraries with coturn server.
2. Some console applications:
    2.1 turnserver.exe: is the main console application to run a TURN/STUN server
    2.2 Other applications: are used to configure or testing purposes.
3. Turnserver.conf: is the configuration file for coturn server.



turnserver.conf

This is the configuration file for coturn server, if you open you will see a default configuration.

Simple Configuration

Your server has the following public IP 80.15.44.123 and listens on port 80. The credentials for connecting are: username = demo, password = secret
Set the following configuration:

```
listening-ip=80.15.44.123
listening-port=80
realm=yourrealm.com
user=demo:secret
```

Configuration with TLS enabled

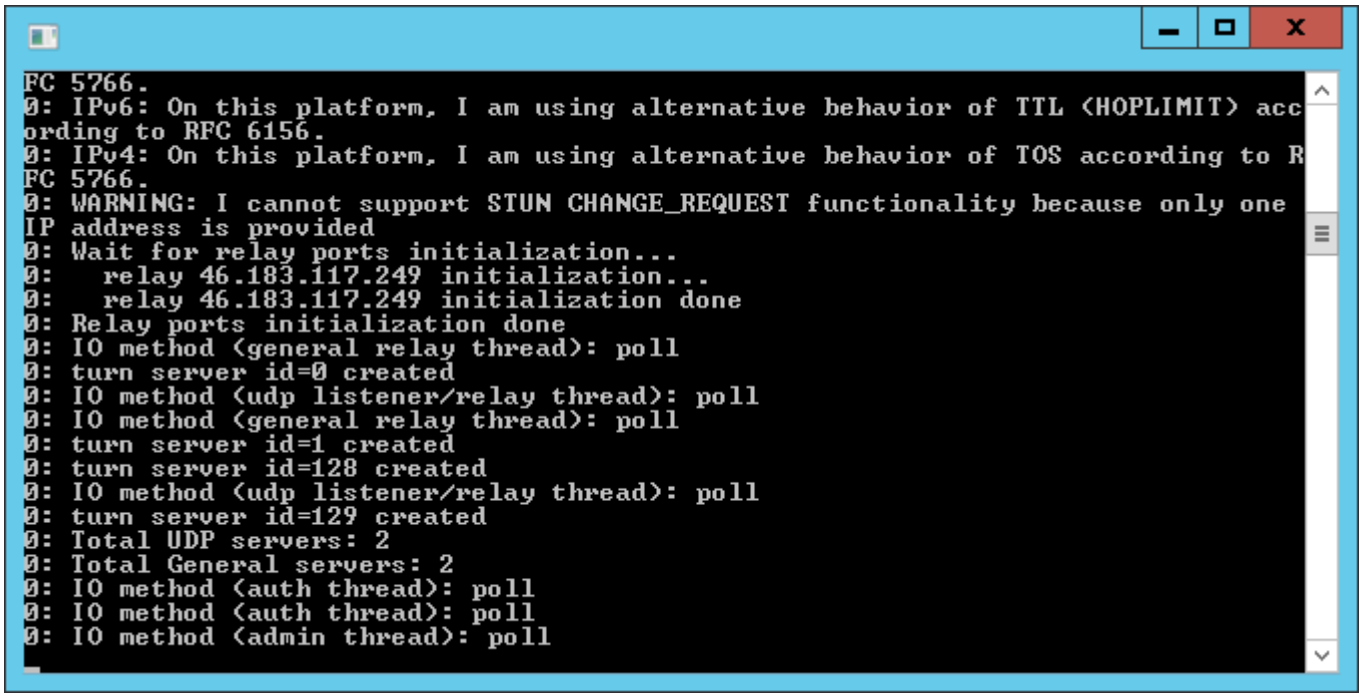
Server has the following public IP 80.15.44.123 and listens on port 80 and 443 (TLS connections). The credentials for connecting are: username = demo, password = secret. Your certificate name (must be in PEM format) is certificate.crt and private key is private.key.
Set the following configuration:

```
listening-ip=80.15.44.123
listening-port=80
realm=yourrealm.com
tls-listening-port=443
cert=certificate.crt
pkey=private.key
user=demo:secret
```

There are more configurations available, just open turnserver.conf and read the documented sections.

Run coturn

Once configured, you can run server just executing turnserver.exe, a new console application will be opened and a log file will be created. You can increase the verbose of console application (get more detailed messages) if you enable "verbose" in turnserver.conf file.



License

eSeGeCe Components End-User License Agreement
eSeGeCe Components ("eSeGeCe") End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and the Author of eSeGeCe for all the eSeGeCe components which may include associated software components, media, printed materials, and "online" or electronic documentation ("eSeGeCe components"). By installing, copying, or otherwise using the eSeGeCe components, you agree to be bound by the terms of this EULA. This license agreement represents the entire agreement concerning the program between you and the Author of eSeGeCe, (referred to as "LICENSER"), and it supersedes any prior proposal, representation, or understanding between the parties. If you do not agree to the terms of this EULA, do not install or use the eSeGeCe components.
The eSeGeCe components are protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The eSeGeCe components are licensed, not sold.
If you want SOURCE CODE you need to pay the registration fee. You must NOT give the license keys and/or the full editions of eSeGeCe (including the DCU editions and Source editions) to any third individuals and/or entities. And you also must NOT use the license keys and/or the full editions of eSeGeCe from any third individuals' and/or entities'.
1. GRANT OF LICENSE
The eSeGeCe components are licensed as follows:
(a) Installation and Use.
LICENSER grants you the right to install and use copies of the eSeGeCe components on your computer running a validly licensed copy of the operating system for which the eSeGeCe components were designed [e.g., Windows 2000, Windows 2003, Windows XP, Windows ME, Windows Vista, Windows 7, Windows 8, Windows 10].
(b) Royalty Free.
You may create commercial applications based on the eSeGeCe components and distribute them with your executables, no royalties required.
(c) Modifications (Source editions only).
You may make modifications, enhancements, derivative works and/or extensions to the licensed SOURCE CODE provided to you under the terms set forth in this license agreement.
(d) Backup Copies.
You may also make copies of the eSeGeCe components as may be necessary for backup and archival purposes.
2. DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS
(a) Maintenance of Copyright Notices.
You must not remove or alter any copyright notices on any and all copies of the eSeGeCe components.
(b) Distribution.
You may not distribute registered copies of the eSeGeCe components to third parties. Evaluation editions available for download from the eSeGeCe official websites may be freely distributed.
You may create components/ActiveX controls/libraries which include the eSeGeCe components for your applications but you must NOT distribute or publish them to third parties.
(c) Prohibition on Distribution of SOURCE CODE (Source editions only).
You must NOT distribute or publish the SOURCE CODE, or any modification, enhancement, derivative works and/or extensions, in SOURCE CODE form to third parties.
You must NOT make any part of the SOURCE CODE be distributed, published, disclosed or otherwise made available to third parties.
(d) Prohibition on Reverse Engineering, Decompilation, and Disassembly.
You may not reverse engineer, decompile, or disassemble the eSeGeCe components, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.
(e) Rental.
You may not rent, lease, or lend the eSeGeCe components.
(f) Support Services.
LICENSER may provide you with support services related to the eSeGeCe components ("Support Services"). Any supplemental software code provided to you as part of the Support Services shall be considered part of the eSeGeCe components and subject to the terms and conditions of this EULA.
eSeGeCe is licensed to be used by only one developer at a time. And the technical support will be provided to only one certain developer.
(g) Compliance with Applicable Laws.
You must comply with all applicable laws regarding use of the eSeGeCe components.
3. ARTIFICIAL INTELLIGENCE (AI) USE
(a) Prohibited Uses.
You may not, directly or indirectly:

(i) use the eSeGeCe components, their source code, or their documentation as training or fine-tuning material for any artificial intelligence or machine learning model, or include them in any dataset, benchmark, or evaluation set used to build, train, or improve such a model;
(ii) transmit the eSeGeCe components, or any part of them, to any third-party AI service that uses the data it receives to train or improve its models, or that retains such data beyond what is transiently required to deliver the service or to comply with applicable law;
(iii) use any AI system to produce reimplementations, ports, functional equivalents, or derivative works of the eSeGeCe components, whether for internal use or for distribution.
(b) Permitted Use of AI Development Assistants.
Subject to the conditions in paragraph (c), you may use AI assistants to support your own development work with the eSeGeCe components. This includes code completion, refactoring assistance, debugging, code explanation, writing integration or glue code, and generating documentation for your own project.
This permission is limited to integrating the eSeGeCe components into your own projects. It does not permit any of the activities prohibited in paragraph (a).
(c) Conditions.
An AI assistant qualifies under paragraph (b) only if one of the following applies:
(i) the provider contractually commits not to use your inputs or outputs to train or improve its models, and not to retain them beyond what is transiently required to deliver the service or to comply with applicable law. Business, enterprise, and API plans carrying such a commitment are examples of qualifying services.
(ii) the model runs locally on hardware you control, so that no part of the eSeGeCe components is transmitted to a third party.
Consumer or free tiers that train on user input, and any service whose terms permit the provider to use submitted content to improve its models, do not qualify. It is your responsibility to verify that the service you use meets these conditions.
(d) Relationship to Other Clauses.
Nothing in this section limits clause 2(c). Output produced by an AI assistant that reproduces the SOURCE CODE of the eSeGeCe components, in whole or in substantial part, remains subject to the prohibition on distributing, publishing, or disclosing SOURCE CODE.
Violation of this section constitutes a material breach of this EULA and results in immediate termination of your license.
4. TERMINATION
Without prejudice to any other rights, LICENSER may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of the eSeGeCe components in your possession.
5. COPYRIGHT
All title, including but not limited to copyrights, in and to the eSeGeCe components and any copies thereof are owned by LICENSER or its suppliers. All title and intellectual property rights in and to the content which may be accessed through use of the eSeGeCe components are the property of the respective content owner and may be protected by applicable copyright or other intellectual property laws and treaties. This EULA grants you no rights to use such content. All rights not expressly granted are reserved by LICENSER.
6. NO WARRANTIES
LICENSER expressly disclaims any warranty for the eSeGeCe components. The eSeGeCe components are provided "As Is" without any express or implied warranty of any kind, including but not limited to any warranties of merchantability, non-infringement, or fitness of a particular purpose. LICENSER does not warrant or assume responsibility for the accuracy or completeness of any information, text, graphics, links or other items contained within the eSeGeCe components. LICENSER makes no warranties respecting any harm that may be caused by the transmission of a computer virus, worm, time bomb, logic bomb, or other such computer program. LICENSER further expressly disclaims any warranty or representation to Authorized Users or to any third party.
7. LIMITATION OF LIABILITY
In no event shall LICENSER be liable for any damages (including, without limitation, lost profits, business interruption, or lost information) rising out of "Authorized Users" use of or inability to use the eSeGeCe components, even if LICENSER has been advised of the possibility of such damages. In no event will LICENSER be liable for loss of data or for indirect, special, incidental, consequential (including lost profit), or other damages based in contract, tort or otherwise. LICENSER shall have no liability with respect to the content of the eSeGeCe components or any part thereof, including but not limited to errors or omissions contained therein, libel, infringements of rights of publicity, privacy, trademark rights, business interruption, personal injury, and loss of privacy, moral rights or the disclosure of confidential information.

Index

A-law 875
AbstractApi Geolocation 1358
Actions 1869
Add Telegram Proxy 751
AddContent 1437
AddSgcHtmlRazor 1418
AE-2 968
AEAD 957
AES 957, 968
AI Data Pack 1410
AI Page Generator 1367
AllowPlainText 213
ALPN 154
Alt-Svc 1022
Amazon SQS 1193
AMQP Channels 388
AMQP Consume Messages 396
AMQP Exchanges 390
AMQP Get Messages 398
AMQP Publish Messages 395
AMQP QoS 399
AMQP Queues 392
AMQP Transactions 400
AMQP1 Links 426
AMQP1 Read Message 434
AMQP1 Receiver Links 430
AMQP1 Send Message 432
AMQP1 Sender Links 427
AMQP1 Sessions 424
Android 96
Anthropic 887
Anthropic Batches 899
Anthropic Citations 906
Anthropic Documents 902
Anthropic Extended Thinking 900
Anthropic Files 911
Anthropic MCP Connector 913
Anthropic Messages 892
Anthropic Models 898
Anthropic Prompt Caching 904
Anthropic Structured Outputs 909
Anthropic Tool Use 896
Anthropic Vision 894
Anthropic Web Search 907
API 3Commas 642
API Binance 496, 512, 519
API Binance Futures 512, 519
API Binance Futures Trade 519
API Bitfinex 611
API Bitget 700
API Bitmex 603
API Blockchain 672
API Bybit 657
API Cex 674
API Cex Plus 682
API Coinbase 522
API Coinbase Pro 522
API Crypto.com 708
API Deribit 705
API Discord 687
API Forex 662
API GateIO 703
API HTX 710
API Kraken 543, 545, 551, 554, 556, 559, 562, 567, 573, 575, 582, 588, 590
API Kraken Futures 573, 575, 582, 588, 590
API Kraken Futures REST Private 590
API Kraken Futures REST Public 588
API Kraken REST Private 556
API Kraken REST Public 554
API Kraken v2 559, 562, 567
API Kraken WebSockets v2 559
API Kucoin 615
API Kucoin Futures 629
API MEXC 692
API MEXC Futures 696
API OKX 647
API OpenAI 690
API Pusher 595
API SignalR 540
API SignalRCore 534
API SocketIO 520

API Telegram 736
API Whatsapp 712
API XTB 654
APIs 493, 496, 504, 507, 512, 519, 520, 522, 527, 530, 534, 540, 543, 545, 551, 554, 556, 559, 562, 567, 573, 575, 582, 588, 590, 595, 603, 611, 642, 662, 669, 672, 674, 687, 692, 696, 736, 1357
APNs 1006, 1007, 1008
  Certificate-Based Connection 1008
  Token-Based Connection 1007
Apple 97
Apple Push Notifications 1004
Argon2 956
asp-for 1427
ASP.NET Core 1418, 1422, 1427, 1430, 1432, 1434
Attributes 1377
Authentication 125, 196, 225, 1390
Authenticator 966
Authorization Code Grant 1121
Authorization Code Grant (RFC 6749) 1091
Authorization Code with PKCE (RFC 7636) 1093
Auto Ban 298
AutoCompleteFor 1430
Backplane 352, 354
barge-in 868, 873
Binance Connect 504
Binance Get Market Data 506
Binance Private Requests Time 510
Binance Private REST API 507
Binance Subscribe 505
Binance Trade Spot 508
Binance Withdraw 511
Binary Message 193
Bindings 134, 208
BIP-340 959
Bitmex Connect WebSocket API 607
Bitmex Subscribe WebSocket Channel 608
BLAKE2 955
BodyEndHTML 1437
Boost 1402
Bot 747
Bot Detection 306
Brainpool 959
Broadcast 133
BroadcastChartAsync 1434
Brute Force 298
Build 80, 81, 83, 84
Build Android Application 83
Build iOS Application 84
Build OSX Application 81
CanDispatchRequest 1385
cbAuto 98
cbNative 98
Certificate 964
Certificate-Based Connection 1008
  APNs 1008
Certificates OpenSSL 185
Certificates SChannel 186
ChaCha20 957
Channels 133, 388, 505, 528, 670, 1262
Chart.js 1437
CheckBoxFor 1427
CIDR 296
Client 176, 178, 179, 191, 192, 193, 196, 199, 200, 201, 202, 231, 371, 373, 374, 386, 387, 405, 406, 480, 995, 996, 998, 999, 1250, 1251, 1252, 1259, 1260, 1261, 1262, 1891, 1892, 1893, 1895, 1900, 1908
Client AMQP Connect 386
Client AMQP Disconnect 387
Client AMQP1 Authentication 421
Client AMQP1 Azure MessageBus 422
Client AMQP1 Connect 417
Client AMQP1 Connection State 420
Client AMQP1 Disconnect 418
Client AMQP1 Idle Timeout Connection 419
Client Authentication 196, 999
Client Certificate Revocation 218
Client Chat 1891
Client Close Connection 178, 995
Client Connections 224
Client Credentials Grant 1123
Client Credentials Grant (RFC 6749) 1095
Client Exceptions 199
Client Kafka Connect 405
Client Kafka Disconnect 406
Client Keep Connection Active 996
Client Keep Connection Open 179
Client MQTT Connect 371

Client MQTT Sessions 373
Client MQTT Version 374
Client Open Connection 176
Client Pending Requests 998
Client Proxies 202
Client Register Protocol 201
Client Send Binary Message 192
Client Send Text 191, 193
Client Send Text Message 191
Client Snapshots 1900
Client SocketIO 1895
Client WebSocket HandShake 200
Clients 373, 386, 405, 406, 480, 996, 998, 1891, 1893
  Send Files 480
Cluster 352, 354, 356
CMAC 955
CMVP 969
Coinbase Pro Connect 527
Coinbase Pro Get Market Data 529
Coinbase Pro Place Orders 532
Coinbase Pro Private Requests Time 531
Coinbase Pro Private REST API 530
Coinbase Pro SandBox Account 533
Coinbase Pro Subscribe 528
ComboBox 1430
CommitOffset 411
CommitSync 409
Common Properties 1377
Composition Layer 1379
Compression 137
Configure Install 63
Configure ZLib 71
Connect Mosquitto 372
Connect Secure Server 184
Connect TCP Server 181
Connect WebSocket Server 175
Connections TIME_WAIT 182
Consume 408
Consumer 408
Consumer Group 409
Content-Encoding 138
Coturn 1910
Crawler Detection 306
CreateTopic 410
CRL 93, 188, 218
Cross-Site Scripting 300
CRUD Pages 1374
Cryptography 952
CryptoHopper 754
CSR 964
CSRF 1390
CSSClass 1377
Custom Objects 139
Custom Sub 124
CustomHead 1437
D3.js 1437
data-sgc-preserve 1402
Datacenter Detection 306
Datasnap 1280
DatePickerFor 1427
DeepSeek 918
DeepSeek Messages 920
DeepSeek Models 922
DeepSeek Vision 921
DefaultSwap 1402
deflate 138
Deflate-Frame 764
DeleteTopic 410
DER 964
DER signature 959
Design in Browser 1365
Design Surface 1363
Design-Time 1363
DevExpress 1418
DevExtreme 1437
Device Authorization Grant (RFC 8628) 1099, 1129
DispatchDirect 1385
DNSBL 306
document extraction 865
DPoP - Demonstrating Proof of Possession (RFC 9449) 1101
DPoP Validation (RFC 9449) 1135
Dropped Disconnections 180
DUnit 1398
ECDH 959
ECDSA 959
ECIES 959
Ed25519 959
Ed448 959
Editions 36

ElementID 1377
Embeddings ChatBot 883
Embeddings Create Vectors 882
Encryption 952
EPOLL 144
Error 242, 260
ErrorToast 1402
EsgcHTMLAssertion 1398
EsgcHTTPAPIProtocolException 493, 506
EventSource 1076
Export 1422
Extended CONNECT 1022
Extensions 762
Fast Performance Server 86
FCrDNS 306
Files 135, 170, 235, 253, 479, 480, 481, 992, 1901
Filtering 1422
FIPS 969
FIPS 140-3 969
FIPS 203 961
FIPS 204 961
FIPS 205 961
FIPS Mode 969
Fired 287
Firewall Blacklist 296
Firewall Whitelist 296
Flood Protection 303
Flow 78
  Threading 78
Forex Account 667
Forex Connect 664, 669
Forex Market Data 668
Forex Subscribe 670
Forex Trade 665
Forward HTTP Requests 155
ForwardedHeaders 309
Found 747
Fragmented Messages 172
function calling 871
Fuzzy Hash 967
G.711 868, 875
GCM 957
Gemini 915
Gemini CountTokens 930
Gemini EmbedContent 931
Gemini Embeddings 931
Gemini Function Calling 932
Gemini Live 868
Gemini Messages 925
Gemini Models 928
Gemini Structured Outputs 929
Gemini Token Counting 930
Gemini Tool Use 932
Gemini Vision 927
Generate 1005
  Remote Notification APNs 1005
GetCommittedOffset 411
GetEarliestOffset 411
GetLatestOffset 411
GetMetadata 410
Getting Started 1361
Google Calendar 1217, 1223, 1224, 1225
Google Calendar RefreshToken 1225
Google Calendar Service Account 1226
Google Calendar Sync Calendars 1223
Google Calendar Sync Events 1224
Google Cloud BigQuery gRPC 1067
Google Cloud FCM 1227
Google Cloud gRPC 1054
Google Cloud Natural Language gRPC 1063
Google Cloud Pub 1208
Google Cloud Pub/Sub gRPC 1055
Google Cloud Speech-to-Text gRPC 1057
Google Cloud Storage gRPC 1065
Google Cloud Translation gRPC 1059
Google Cloud Vision gRPC 1061
Google OAuth2 Keys 1198
Google Service Accounts 1204
Google Vertex AI gRPC 1069
Grid 1422
Grok 940
Grok Messages 942
Grok Models 944
Grok Vision 943
GroupId 409
Groups 140
gRPC 1047
gzip 138
Hashing 952
HeartBeat 129
Hello World 1361

HighPerfAcceptsPerWorker 278
History 162, 667
HKDF 956
HMAC 955
HOTP 966
How to Place a Bitmex Order 609
htAuto 1406
HTML Components 1359
Html.Sgc 1418, 1422
Html.Sgc().Chart 1434
Html.Sgc().Dialog 1432
htSSE 1406
HTTP 76, 132, 155, 234, 235, 242, 243, 252, 253, 260, 261, 282, 986, 990
HTTP Compression 138
HTTP Dispatch Files 235, 253
HTTP Server Stream Video 246, 264
HTTP/2 236, 237, 239, 254, 255, 257, 991, 992, 993, 994, 997, 1000
HTTP/2 Alternate Service 239, 257
HTTP/2 Download File 992
HTTP/2 Headers 994
HTTP/2 Partial Responses 993
HTTP/2 Reason Disconnection 997
HTTP/2 Server Push 237, 255, 991
HTTP/2 Server Threads 240, 258
HTTP/3 1022
HTTP1 1071
HTTP2 987
HTTP3 1022
HTTPAPI 280, 282, 283, 287
HTTPAPI Custom Headers 283
HTTPAPI Disable HTTP 282
HTTPAPI FineTune 278
HTTPAPI OnDisconnect 287
HTTPAPI Send File Response 285
HTTPAPI Send Text Response 284
HTTPAPI URL Reservation 276
ICE 1267
ICE Gather Candidates 1269
ICE Pair Candidates 1270
Identity Assertion Authorization Grant 1105
In HTML 1115
Indy 117
Indy SChannel 215, 247, 265
Install Errors 59
Install Package 52
Install Setup 41
Install sgcIndy Package 65
Installation 38
Instruments 1450
Introduction 31
invoice 865
IOCP 142
iohNativeTLS 98, 149
IoT 972, 973, 981
IoT Amazon MQTT Client 973
IoT Azure MQTT Client 981
IP Ban 298
IP Filtering 296
IPv6 964
ISgcHtmlHub 1434
JavaScript 1437
JWT 1144
Kafka 405, 406, 407, 408, 409, 410, 411
Kafka Consume Messages 408
Kafka Consumer Groups 409
Kafka Offsets 411
Kafka Produce Messages 407
Kafka Topics 410
KendoUI_Grid 1904
Key Wrap 957
Kraken v2 Trading 567
Kucoin Connect WebSocket API 622
Kucoin Futures Connect WebSocket API 634
Kucoin Futures Get Market Data 636
Kucoin Futures Private Requests Time 641
Kucoin Futures Private REST API 637
Kucoin Futures Subscribe WebSocket Channel 635
Kucoin Futures Trade 638
Kucoin Get Market Data 624
Kucoin Private Requests Time 628
Kucoin Private REST API 625
Kucoin Subscribe WebSocket Channel 623
Kucoin Trade Spot 626
Last-Event-ID 1079, 1406
License 1912
Lightstreamer 669, 670
Live Chart 1434
Live Preview 1365
LoadBalancing 169

LoadingIndicator 1402
Locality Sensitive Hashing 967
Login 1390
Logout 1390
Logs 131
Lookup 1430
MaxConnectionsPerIP 303
MaxMessagesPerSec 303
MCP 765, 871
MCP Apps 1371
MCP Client Authorization;OAuth;PKCE;Dynamic Client Registration;MCPOptions.Authorization 817
MCP Client Elicitation 811
MCP Client Multi Round-Trip Requests;MRTR;input_required;OnMCPInputR 813
MCP Client OAuth2 Authentication 815
MCP Client Prompts 803
MCP Client Protocol Versions;ProtocolEra;server/discover;2026-07-28 796
MCP Client Resources 805
MCP Client Roots 807
MCP Client Sampling 809
MCP Client Subscriptions;subscriptions/listen;SubscriptionsListen 798
MCP Client Tasks;io.modelcontextprotocol/tasks;TasksGet;OnMCPTaskCompleted 799
MCP Client Tools 801
MCP Server Elicitation 791
MCP Server Logging and Progress;notifications/message;notifications/progress 776
MCP Server Multi Round-Trip Requests;MRTR;input_required 793
MCP Server Prompts 781
MCP Server Protocol Versions;MCP 2026-07-28;server/discover 771
MCP Server Resources 784
MCP Server Roots 787
MCP Server Sampling 789
MCP Server Sessions 770
MCP Server Subscriptions;subscriptions/listen 773
MCP Server Tasks;io.modelcontextprotocol/tasks 774
MCP Server Tools 777
MCPEndpoint 1371
Memory Manager 89
Mesh 352
Message Filtering 300
Message History 162
  Reconnect Recovery 162
Method 990
Minimal API 1418
Mistral AI 945
Mistral Embeddings 951
Mistral Messages 948
Mistral Models 950
Mistral Vision 949
ML-DSA 961, 964, 1144
ML-DSA-44 1144
ML-DSA-65 1144
ML-DSA-87 1144
ML-KEM 961, 964
Modal 1432
morph 1402
MQTT 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 973, 981, 1893
MQTT Clear Retained Messages 381
MQTT Publish 375, 378, 380
MQTT Publish Message 378
MQTT Publish Subscribe 375
MQTT Receive Messages 379
MQTT Subscribe 377
MQTT Topics 376
mu-law 875
Multi-Node 352, 354, 356
Mutual TLS 218
MVC 1418
Name Constraints 964
Native TLS 98, 149
NativeTLS_Options 149
Navigation 1402
Near Duplicate 967
NIST 969
Notification Requests 1006
  Sending 1006
OAuth2 1000, 1082, 1111, 1115, 1116, 1117, 1118, 1119, 1120, 1198

OAuth2 Client for Desktop Applications 1088
OAuth2 Client for Web Applications 1087
OAuth2 Customize Sign 1115
OAuth2 None Authenticate URLs 1120
OAuth2 Provider Authentication 1141
OAuth2 Provider Azure AD 1139
OAuth2 Provider Private Endpoints 1140
OAuth2 Provider Requests 1143
OAuth2 Recover Access Tokens 1118
OAuth2 Register Apps 1117
OCR 863, 865
OCSP 93, 188, 218
Offset 162
Ollama 934
Ollama Embeddings 939
Ollama Messages 937
Ollama Models 938
ompClassic 278
ompHighPerf 278
OnExecute 1869
OnToolCall 871
OpenAI 819
OpenAI Applications 838
OpenAI Audio 830, 847
OpenAI Batch 836
OpenAI Chat 828
OpenAI Completion 827
OpenAI Edit 829
OpenAI Fine-Tuning 835
OpenAI Moderation 831
OpenAI Realtime 868
OpenAI RealTime 832
OpenAI Responses 833
OpenAI Speech 834
OpenAI Uploads 837
OpenAPI 1287
OpenAPI Additional Properties 1295
OpenAPI Amazon AWS 1317
OpenAPI Amazon AWS Credentials 1323
OpenAPI Amazon AWS S3 1325
OpenAPI API 1302
OpenAPI Client 1303
OpenAPI Command Line 1297
OpenAPI Google Cloud 1326
OpenAPI Google Cloud Calendar 1340
OpenAPI Google Cloud OAuth2 1331
OpenAPI Google Cloud PubSub 1339
OpenAPI Google Cloud Service Accounts 1334
OpenAPI Google Drive 1341
OpenAPI Library 1300
OpenAPI Microsoft 1343
OpenAPI Microsoft Graph 1356
OpenAPI Microsoft OAuth2 Code 1352
OpenAPI Microsoft OAuth2 Credentials 1354
OpenAPI Microsoft Register Application 1349
OpenAPI Microsoft Tenant 1348
OpenAPI Parser Pascal 1289
OpenAPI Server 1307
OpenAPI Server Code-First 1312
OpenAPI Server Spec-First 1310
OpenSSL 98, 100, 103, 105, 107, 108, 116, 185
OpenSSL Android 107
OpenSSL iOS 108
OpenSSL Load Additional Functions 114
OpenSSL OSX 105
OpenSSL Own CA Certificates 110
OpenSSL P12 Certificates 112
OpenSSL Verify Certificate 113
OpenSSL Windows 103
OperatingMode 278
Orders 665, 667
Overview 72
PagePath 1869
Paging 1422
Passkeys 1151
Password Hashing 956
Path Length 964
Path Parameters 1385
PBKDF2 956
PCMA 875
PCMU 875
PDF-Back-Cover 1929
PDF-Front-Cover 1
Peers 352
PEM 964
PerMessage-Deflate 763
PFX 149
Pinecone 884
PKCS#10 964

PKCS#12 149
PKCS#8 959, 961
PlainTextPolicy 213
PlainTextTimeout 213
Poll 408
Polling 1434
Poly1305 955
Popup Editing 1432
Positions 667
Post Big Files 135
Post-Quantum 149, 961, 1144
Post-Quantum Certificate 964
Presence 356
Price Bars 668
Price Ticks 668
Produce 407
Producer 407
Protocol AMQP 382
Protocol AMQP1 412
Protocol Dataset 466, 470, 473, 474
Protocol Dataset Javascript 470
Protocol Dataset Notify Updates 474
Protocol Dataset Replicate Table 473
Protocol Default 457, 462
Protocol Default Javascript 462
Protocol E2EE 489
Protocol Files 475
Protocol Kafka 402
Protocol MQTT 368
Protocol Presence 482, 486
Protocol Presence Javascript 486
Protocol STOMP 435
Protocol WAMP 442, 445
Protocol WAMP Javascript 445
Protocol WAMP2 453
Protocol WebRTC 439, 441
Protocol WebRTC Javascript 441
Protocols 124, 201, 363, 365, 368, 382, 402, 435, 439, 441, 442, 445, 453, 457, 462, 466, 470, 473, 474, 475, 482, 486
Protocols Javascript 365
Proxy 171, 202, 751
Pub/Sub 354
push to talk 873
PushChartAsync 1434
PushURL 1402
QPACK 1022
Quality 156
  Service 156
Quality Of Service 156
QUERY 1022
QueueLength 278
Queues 158, 392
QUIC 1010
QUIC engine 1010
QUIC streams 1010
QuickStart 1361
QuickStart HTTP 76
QuickStart WebSockets 74
Rate Limiting 303
Razor 1418, 1422, 1427, 1430, 1432, 1434
RCON 753
Real Client IP 309
realtime 868
Receive Binary Messages 195, 230
Receive Text Messages 194, 229
Reconnect Recovery 162
ReconnectUI 1402, 1879
Recovery 162
Redis 354
Refresh Token Grant 1127
Register 752
Register Telegram User 752
Remember Me 1390
Remote Notification APNs 1005
  Generate 1005
Replay 162
Request HTTP 990
RequireLogin 1390
RequireRoles 1390
Resource Owner Password Credentials Grant 1125
Resource Owner Password Credentials Grant (RFC 6749) 1097
Response Body 242, 260
REST API 664
Reverse DNS 306
Reverse Proxy 309
Revocation 93, 188, 218
RFC 10024 961
RFC 4226 966
RFC 6238 966

RFC 9000 1010
RFC 9001 1010
RFC 9002 1010
RFC 9114 1022
RFC 9204 1022
RFC 9881 961
RFC 9909 961
RFC 9935 961
RFC 9964 1144
Right-to-left 1382
Routing 1385
RSA 959
RTCPeerConnection Data 1279
RTCPeerConnection DTLS 1278
RTCPeerConnection ICE 1277
RTCPeerConnection Signaling 1276
RTCPeerConnection STUN TURN 1275
RTCPeerConnection WebSocket Client 1274
RTCPeerConnection WebSocket Server 1273
RTL 1382
Runtime 1363
Salsa20 957
SChannel 93, 215, 247, 265
SChannel Certificate Revocation 188
SChannel Get Connection Info 190
SChannel Server 215, 247, 265
Schnorr 959
ScrollRestoration 1402
scrypt 956
secp256k1 959
Secure Connections 127
SelectFor 1427
Self-Signed Certificates 281
Send Big Files 481
Send Files 479, 480
  Clients 480
  Server 479
Send Files To Clients 480
Send Files To Server 479
Send Telegram Invoice Message 749
Send Telegram Message Bold 746
Send Telegram Message With Buttons 744, 745
Send Telegram Message With Inline Buttons 744
Sending 1006
  Notification Requests 1006
Server 207, 208, 209, 210, 211, 221, 222, 223, 225, 227, 228, 229, 230, 231, 234, 236, 243, 252, 254, 261, 280, 372, 479, 991, 1111, 1116, 1119, 1254, 1255, 1265, 1266, 1889, 1896, 1899, 1903, 1907
  Send Files 479
Server Authentication 225, 1119, 1903
Server Bindings 208
Server Chat 1889
Server Close Connection 223
Server Endpoints 1116
Server Example 1111
Server Keep Active 210
Server Keep Connections Alive 221
Server Monitor 1896
Server Plain TCP 222
Server Read Headers 231
Server Requests 234, 252
Server Send Binary Message 228
Server Send Text Message 227
Server Sessions 243, 261
Server Snapshots 1899
Server SSL 211, 213, 280
Server SSL SChannel 215, 247, 265
Server SSL SChannel Client Certificate Revocation 218
Server Start 207
Server Startup Shutdown 209
Server Verify Certificate 220
Server-Sent Events 167, 1076, 1406
ServerSentEvents 1906
Service 156, 239, 257, 1204
  Quality 156
Session Resumption 149
Sessions 1390
SessionTickets 149
SGC_CRYPTO_FIPS 969
SGC_NATIVE_CRYPTO 98
SGC_UPDATE_SNAPSHOTS 1398
sgc-autocomplete 1430
sgc-chart 1434
sgc-checkbox 1427
sgc-column 1422
sgc-datepicker 1427
sgc-dialog 1432

sgc-form 1427
sgc-grid 1422
sgc-scripts 1418
sgc-select 1427
sgc-series 1434
sgc-styles 1418
sgc-submit 1427
sgc-textarea 1427
sgc-textbox 1427
SgcAutoComplete 1430
SgcChart 1434
SgcChartBuilder 1434
SgcChartSeries 1434
SgcCloseDialog 1432
sgcCrypto 98, 952
sgcCryptoBackend 98
SgcDataRequest 1422
SgcDataResult 1422
SgcExportFormat 1422
SgcGrid 1422
SgcGridExport 1422
SgcGridFilterKind 1422
sgcHTML 1359, 1361, 1363, 1365, 1385, 1390, 1398, 1402, 1406, 1418, 1422, 1427, 1430, 1432, 1434, 1437
sgcHTML Localization 1382
sgcHTML Nodes 1379
sgcHTML pages on DataSnap Servers 1285
sgcHTML_Testing 1398
sgcHTMLGetDirection 1382
sgcHTMLIsRTLLanguage 1382
sgcHTMLNewToken 1824
sgcHTMLPasswordHash 1819
sgcHTMLPasswordVerify 1819
sgcHTMLRequestAuth 1819
sgcHTMLRequestSession 1824
sgcHTMLSetIDSeed 1398
sgcHTMLSetLanguage 1382
sgcHTMXGetAsset 1402
sgcIdSSLOpenSSL_Static 116
SgcItem 1430
sgcProtoBuf 1048
SgcRedirect 1427
SgcResults 1422
SgcResults.AutoComplete 1430
SgcResults.Chart 1434
SgcResults.CloseDialog 1432
SHA-256 955
SHA-3 955
SHAKE 955
SIP 875
SipHash 955
SkipIOCPOnSuccess 278
SLH-DSA 961, 964
Snapshots 1398
Sorting 1422
SPA 1402
SQL Injection 300
SSE 1076, 1079, 1081, 1406
SSEEndpoint 1879
SSL 92, 93, 96, 97
Static Linking 116
Stop Limit 665
STUN 1247, 1250, 1251, 1252, 1254, 1255
STUN Client Attributes 1252
STUN Client Long Term Credentials 1251
STUN Client UDP Retransmissions 1250
STUN Server Alternate Server 1255
STUN Server Long Term Credentials 1254
Style 1377
Sub 1208
SubjectPublicKeyInfo 961
SubProtocol 165
Subscribe 408
SubscribeStreams 505
Tag Helpers 1418
Taproot 959
TCP Connections 164
Telegram Chat 747
Telegram Client 1908
Telegram Get SuperGroup Members 750
Telegram Sponsored Messages 748
telephony 875
Telerik 1418
Testing 1398
TextAreaFor 1427
TextBoxFor 1427
third-party libraries 1437
Threading 78
  Flow 78
Throttle 166, 493
TicketLifetime 149

TLS 92, 93, 96, 97
TLS 1.2 149
TLS 1.3 149
TLS and plain HTTP on the same port 213
TLS Provider 146
TLSH 967
Token Introspection (RFC 7662) 1133
Token Revocation (RFC 7009) 1131
Token-Based Connection 1007
  APNs 1007
tools 871
ToSgcDataResult 1422
TOTP 966
Transactions 160
Transport 1406, 1879
TrustedHops 309
TrustedProxies 309
TsgcAIChat 853
TsgcAIChat Documents and PDF 857
TsgcAIChat Structured Output 854
TsgcAIDatabaseVectorFile 880
TsgcAIDatabaseVectorPinecone 881
TsgcAIDocumentExtractor 859, 865
TsgcAIOCRTesseract 861
TsgcAIOCRWindows 863
TsgcAIOpenAIAssistant 839
TsgcAIOpenAIAssistant File Search 841
TsgcAIOpenAIAssistant Function Calling 845
TsgcAIOpenAIAssistant Streaming 843
TsgcAIOpenAIChatBot 877
TsgcAIOpenAIEmbeddings 879
TsgcAIOpenAITranslator 878
TsgcAIVoiceAgent 868, 871, 873, 875
TsgcAudioPlayerMCI 849
TsgcAudioRecorderMCI 848
TsgcBotClassification 306
TsgcGRPCClient 1050
TsgcHTMLAIAssistant 1808
TsgcHTMLAIColumns 1814
TsgcHTMLAnnunciatorTile 1468
TsgcHTMLAuth 1819
TsgcHTMLAuthRateLimit_Options 1819
TsgcHTMLComponent 1377
TsgcHTMLComponent_Accordion 1712
TsgcHTMLComponent_ActivityFeed 1785
TsgcHTMLComponent_AIChat 1805
TsgcHTMLComponent_AnalogClock 1473
TsgcHTMLComponent_Annunciator 1465
TsgcHTMLComponent_AuditTrail 1621
TsgcHTMLComponent_AutoComplete 1645
TsgcHTMLComponent_Avatar 1719
TsgcHTMLComponent_Badge 1767
TsgcHTMLComponent_Barcode 1533
TsgcHTMLComponent_Breadcrumb 1548
TsgcHTMLComponent_ButtonGroup 1566
TsgcHTMLComponent_Calendar 1590
TsgcHTMLComponent_CameraScanner 1696
TsgcHTMLComponent_CandlestickChart 1525
TsgcHTMLComponent_Carousel 1714
TsgcHTMLComponent_Chart 1440
TsgcHTMLComponent_Chat 1802
TsgcHTMLComponent_ChatBox 1799
TsgcHTMLComponent_CheckBox 1635
TsgcHTMLComponent_Chip 1770
TsgcHTMLComponent_ColorPicker 1684
TsgcHTMLComponent_CommandPalette 1776
TsgcHTMLComponent_Compass 1476
TsgcHTMLComponent_ContextMenu 1773
TsgcHTMLComponent_DataTable 1586
TsgcHTMLComponent_DatePicker 1649
TsgcHTMLComponent_DateRangePicker 1674
TsgcHTMLComponent_DateTimePicker 1671
TsgcHTMLComponent_Diagram 1510
TsgcHTMLComponent_Dropdown 1563
TsgcHTMLComponent_Edit 1629
TsgcHTMLComponent_EmptyState 1760
TsgcHTMLComponent_FileUpload 1652
TsgcHTMLComponent_Form 1625
TsgcHTMLComponent_Gantt 1603
TsgcHTMLComponent_Gauge 1445
TsgcHTMLComponent_Geolocation 1701
TsgcHTMLComponent_Grid 1575
TsgcHTMLComponent_Heatmap 1517
TsgcHTMLComponent_Image 1717
TsgcHTMLComponent_ImpersonateBanner 1847
TsgcHTMLComponent_InlineAIPrompt 1811
TsgcHTMLComponent_InputGroup 1643
TsgcHTMLComponent_JobProgress 1789

TsgcHTMLComponent_KanbanBoard 1598
TsgcHTMLComponent_Knob 1479
TsgcHTMLComponent_LED 1456
TsgcHTMLComponent_LEDBar 1459
TsgcHTMLComponent_LinearGauge 1453
TsgcHTMLComponent_ListGroup 1569
TsgcHTMLComponent_Login 1829
TsgcHTMLComponent_LogViewer 1794
TsgcHTMLComponent_Map 1512
TsgcHTMLComponent_Memo 1632
TsgcHTMLComponent_Modal 1733
TsgcHTMLComponent_MultiSelect 1661
TsgcHTMLComponent_NavBar 1536
TsgcHTMLComponent_Notification 1746
TsgcHTMLComponent_NotificationInbox 1750
TsgcHTMLComponent_NotificationPreference 1751
TsgcHTMLComponent_NumericStepper 1488
TsgcHTMLComponent_NumPad 1691
TsgcHTMLComponent_OAuthCallback 1836
TsgcHTMLComponent_Odometer 1469
TsgcHTMLComponent_Offcanvas 1736
TsgcHTMLComponent_Oscilloscope 1494
TsgcHTMLComponent_Pagination 1553
TsgcHTMLComponent_Panel 1706
TsgcHTMLComponent_PDFViewer 1728
TsgcHTMLComponent_PivotTable 1608
TsgcHTMLComponent_Placeholder 1757
TsgcHTMLComponent_Popover 1738
TsgcHTMLComponent_Presence 1781
TsgcHTMLComponent_ProgressBar 1764
TsgcHTMLComponent_PushButton 1485
TsgcHTMLComponent_QRCode 1530
TsgcHTMLComponent_RadioGroup 1637
TsgcHTMLComponent_RangeSlider 1681
TsgcHTMLComponent_Rating 1658
TsgcHTMLComponent_RichEditor 1655
TsgcHTMLComponent_RolesPermissions 1850
TsgcHTMLComponent_SCADAPanel 1497
TsgcHTMLComponent_Scheduler 1593
TsgcHTMLComponent_SegmentDisplay 1462
TsgcHTMLComponent_Select 1640
TsgcHTMLComponent_Sidebar 1539
TsgcHTMLComponent_SignaturePad 1687
TsgcHTMLComponent_Site 1542
TsgcHTMLComponent_Slider 1677
TsgcHTMLComponent_Snackbar 1743
TsgcHTMLComponent_SocialLogin 1833
TsgcHTMLComponent_Sparkline 1514
TsgcHTMLComponent_Spinner 1754
TsgcHTMLComponent_Splitter 1725
TsgcHTMLComponent_StatCard 1709
TsgcHTMLComponent_Stepper 1560
TsgcHTMLComponent_StripChart 1491
TsgcHTMLComponent_Tabs 1550
TsgcHTMLComponent_Timeline 1595
TsgcHTMLComponent_TimePicker 1668
TsgcHTMLComponent_Toast 1741
TsgcHTMLComponent_ToggleSwitch 1482
TsgcHTMLComponent_Toolbar 1556
TsgcHTMLComponent_Transfer 1664
TsgcHTMLComponent_TreeGrid 1617
TsgcHTMLComponent_TreeMap 1521
TsgcHTMLComponent_TreeView 1558
TsgcHTMLComponent_UserManagement 1842
TsgcHTMLComponent_Video 1722
TsgcHTMLComponent_VUMeter 1471
TsgcHTMLComponent_WebAuthnLogin 1839
TsgcHTMLContainer 1379
TsgcHTMLDashboardLayout 1572
TsgcHTMLDesignAction 1869
TsgcHTMLDesignServer 1361, 1365, 1869
TsgcHTMLDesignSurface 1361, 1363, 1365, 1869
TsgcHTMLDOMLite 1398
TsgcHTMLEngine_Server 1855
TsgcHTMLGridOps 1816
TsgcHTMLInstrumentsBinder 1502
TsgcHTMLInstrumentsMQTTBinder 1505
TsgcHTMLLanguage 1382
TsgcHTMLLocale 1382
TsgcHTMLNode 1379
TsgcHTMLPageBuilder 1873
TsgcHTMLPivotOps 1818
TsgcHTMLSession 1390, 1824
TsgcHTMLSessionStore 1824
TsgcHTMLSessionStore_File 1827

TsgcHTMLSessionStore_Memory 1824
TsgcHTMLTemplate_Bootstrap 1875
TsgcHTMLTemplate_HTMX 1879
TsgcHTMLThemeBuilder 1883
TsgcHTMLThemeController 1881
TsgcHTMLWebPush 1752
TsgcHTMX_Engine_Server 1857
TsgcHTMX_Engine_Server_HTTPAPI_WebBroke 1866
TsgcHTMX_Engine_Server_WebBroker 1863
TsgcHTMX_Engine_Server_WebBrokerBridge 1860
TsgcHTMX_Fragment 1885
TsgcHTMX_Route 1385
TsgcHTMX_Router 1887
TsgcHTMXDispatchRequest 1385
TsgcHTMXDispatchResponse 1385
TsgcHTMXRequest 1385
TsgcHTMXTransport 1879
TsgcHTTP_API_Anthropic 891
TsgcHTTP_API_DeepSeek 919
TsgcHTTP_API_Gemini 917
TsgcHTTP_API_Grok 941
TsgcHTTP_API_Mistral 947
TsgcHTTP_API_Ollama 936
TsgcHTTP_API_OpenAI 826
TsgcHTTP_JWT_Client 1146
TsgcHTTP_JWT_Server 1149
TsgcHTTP_OAuth2_Client 1083
TsgcHTTP_OAuth2_Client_Google 1089
TsgcHTTP_OAuth2_Client_Microsoft 1090
TsgcHTTP_OAuth2_Server 1108
TsgcHTTP_OAuth2_Server_Provider 1137
TsgcHTTP_OIDC_Client 1186
TsgcHTTP2Client 988
TsgcHTTP2ConnectionClient 1001
TsgcHTTP2RequestProperty 1002
TsgcHTTP2ResponseProperty 1003
TsgcHTTP3Client 1026
TsgcHTTP3Server 1031
TsgcHTTPRESTServer 268
TsgcHTTPServer 250
TsgcHTTPServer_Sessions 261
TsgcHTTPServer_Tenancy 270
TsgcHTTPServer_Users 269
TsgcHTTPServerStats 272
TsgcICEClient 1268
TsgcIWWebSocketClient 359
TsgcIWWSPClient_Dataset 469
TsgcIWWSPClient_sgc 461
TsgcLDAPClient 1180
TsgcMailOAuth2 1191
TsgcOIDCJWKS 1189
TsgcQUICClient 1014
TsgcQUICServer 1018
TsgcRTCPeerConnection 1271
TsgcSAMLResult 1185
TsgcSAMLServiceProvider 1182
TsgcServerHTTPAPI_FineTune 278
TsgcSSEClient 1077, 1079
TsgcSSEEvent 1080
TsgcSSEParser 1081
TsgcSTUNClient 1248
TsgcSTUNServer 1253
TsgcTCPClient 203
TsgcTextToSpeechAmazon 852
TsgcTextToSpeechGoogle 851
TsgcTextToSpeechSystem 850
TsgcTOTPAuthenticator 1178
TsgcTURNClient 1257
TsgcTURNServer 1263
TsgcUDPClient 1245
TsgcUDPServer 1246
TsgcWebPush_Client 761
TsgcWebSocketClient 173
TsgcWebSocketClient_WinHTTP 288
TsgcWebSocketFirewall 289, 296, 298, 300, 303, 306, 309
TsgcWebSocketHTTPServer 232
TsgcWebSocketHTTPServer_Sessions 243
TsgcWebSocketLoadBalancerServer 357
TsgcWebSocketProxyServer 358
TsgcWebSocketServer 205
TsgcWebSocketServer_HTTPAPI 274
TsgcWebTransportSession 1039
TsgcWebTransportStream 1043
TsgcWebView2 1230
TsgcWSAPIClient_MCP 795
TsgcWSAPIKeyManager 336
TsgcWSAPIServer_MCP 767
TsgcWSAPIServer_WebAuthn 1154
TsgcWSAPIServer_WebPush 760

TsgcWSCircuitBreaker 325
TsgcWSCluster 348, 352, 354, 356
TsgcWSConnection 361
TsgcWSHTTP2WebBrokerBridgeServer 1283
TsgcWSHTTPWebBrokerBridgeServer 1281
TsgcWSMessageFile 478
TsgcWSPClient_AMQP 383
TsgcWSPClient_AMQP1 414
TsgcWSPClient_Dataset 468
TsgcWSPClient_E2EE 492
TsgcWSPClient_Files 477
TsgcWSPClient_Kafka 403
TsgcWSPClient_MQTT 370
TsgcWSPClient_Presence 485
TsgcWSPClient_sgc 460
TsgcWSPClient_STOMP 436
TsgcWSPClient_STOMP_ActiveMQ 438
TsgcWSPClient_STOMP_RabbitMQ 437
TsgcWSPClient_WAMP 444
TsgcWSPClient_WAMP2 454
TsgcWSPresenceMessage 484
TsgcWSPServer_Dataset 467
TsgcWSPServer_E2EE 491
TsgcWSPServer_Files 476
TsgcWSPServer_Presence 356, 483
TsgcWSPServer_sgc 459
TsgcWSPServer_WAMP 443
TsgcWSPServer_WebRTC 440
TsgcWSRateLimiter 313
TsgcWSServer_HTTPAPI_WebBrokerBridge 1284
TURN 1256, 1259, 1260, 1261, 1262, 1265, 1266
TURN Client Allocate IP Address 1259
TURN Client Create Permissions 1260
TURN Client Send Indication 1261
turn detection 873
TURN Server Allocations 1266
TURN Server Long Term Credentials 1265
Upload File 1901
UseSystemRoots 149
Using DLL 121
VAD 873
Validation 1427
ViewTransitions 1402
voice agent 868
Wait Response 380
WAMP 448, 449, 450, 451
WAMP Publishers 449
WAMP RPC Progress Results 451
WAMP Simple RPC 450
WAMP Subscribers 448
WatchDog 130
Web Browser Test 123
Web Design 1365
Web Push 1415
WebAuthn 1150
WebAuthn Authentication 1165
WebAuthn Authentication Request 1167
WebAuthn Authentication Response 1168
WebAuthn Authentication Result 1169
WebAuthn Authorization 1171
WebAuthn Authorization HTTP 1172
WebAuthn Authorization WebSocket 1173
Webauthn Javascript Client 1174
WebAuthn MDS 1170
WebAuthn Registration 1158
WebAuthn Registration Request 1161
WebAuthn Registration Response 1162
WebAuthn Registration Result 1164
WebPush 759
WebRTC 1907
WebSocket Events 119
WebSocket Parameters Connection 120
WebSocket Redirections 183
WebSockets 74, 119, 120, 175, 183, 200, 504, 505, 527, 528, 545, 551, 562, 567, 575, 582, 669, 670
WebSockets Private 551, 567, 582
WebSockets Public 545, 562, 575
WebTransport 1022, 1035
WebView2 1365
WebView2 Advanced Features 1242
WebView2 Cookies 1236
WebView2 Downloads 1238
WebView2 JavaScript 1234
WebView2 Navigation 1232
WebView2 Settings 1240
WhatsApp Create App 716
WhatsApp Download Media 735
WhatsApp Phone Number Id 718

WhatsApp Receive Messages and Status Notifications 731
WhatsApp Security 721
WhatsApp Send Files 733
WhatsApp Send Interactive Messages 725
WhatsApp Send Messages 722
WhatsApp Send Template Messages 729
WhatsApp Token 719
WhatsApp Webhook 720
Window 1432
Windows OCR 863
Windows Service 122
WinZip 968
Without OpenSSL 98
WriteSSEEvent 1406
X-Forwarded-For 309
X-Real-IP 309
X-SGC-Connection 1406
X-Wing 961
X.509 964
X25519 959
X25519MLKEM768 149, 961
X448 959
XSS 300
Zero Code 1361, 1869
Zip Encryption 968

Copyright © 2012-2026 eSeGeCe Software
info@esegece.com
www.esegece.com