sgcWebSockets 2026.5

May 2026

Documentation for Delphi

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

Contents

Introduction **23**

Overview **28**

Editions 28

Installation 30

Install **33**

Install Setup 33

Install Package 44

Install Errors 51

Configure Install 55

Install sgcIndy Package 57

Configure ZLib 63

QuickStart **64**

Overview 64

QuickStart WebSockets 66

QuickStart HTTP 68

Threading Flow 70

Build 72

Build OSX Application 73

Build Android Application 75

Build iOS Application 76

Fast Performance Server 78

Memory Manager 81

OpenSSL 84

OpenSSL Windows 87

OpenSSL OSX 89

OpenSSL Android 91
OpenSSL iOS 92
OpenSSL Own CA Certificates 94
OpenSSL P12 Certificates 96
OpenSSL Verify Certificate 97
OpenSSL Load Additional Functions 98
Indy 99

Topics 101

WebSocket Events 101
WebSocket Parameters Connection 102
Using inside a DLL 103
Windows Service 104
Web Browser Test 105
Custom Sub-Protocols 106
Authentication 109
Secure Connections 111
HeartBeat 113
WatchDog 114
Logs 115
HTTP 116
Broadcast and Channels 117
Bindings 118
Post Big Files 119
Compression 121
Flash 122
Custom Objects 123
Groups 124
IOCP 126
EPOLL 127
ALPN 128

Forward HTTP Requests 129
Quality Of Service 130
Queues 132
Transactions 134
TCP Connections 136
SubProtocol 137
Throttle 138
Server-Sent Events 139
LoadBalancing 141
Files 142
Proxy 143
Fragmented Messages 144

Components 145

TsgcWebSocketClient 145
Connect WebSocket Server 147
Client Open Connection 148
Client Close Connection 150
Client Keep Connection Open 151
Dropped Disconnections 152
Connect TCP Server 153
Connections TIME_WAIT 154
WebSocket Redirections 155
Connect Secure Server 156
Certificates OpenSSL 157
Certificates SChannel 158
SChannel Get Connection Info 160
Client Send Text Message 161
Client Send Binary Message 162
Client Send Text and Binary Message 163
Receive Text Messages 164

Receive Binary Messages 165
Client Authentication 166
Client Exceptions 169
Client WebSocket HandShake 170
Client Register Protocol 171
Client Proxies 172
TsgcTCPClient 173
TsgcTCPClient 174
TsgcWebSocketServer 175
Server Start 177
Server Bindings 178
Server Startup Shutdown 179
Server Keep Active 180
Server SSL 181
Server SSL SChannel 183
Server Verify Certificate 186
Server Keep Connections Alive 187
Server Plain TCP 188
Server Close Connection 189
Client Connections 190
Server Authentication 191
Server Send Text Message 192
Server Send Binary Message 193
Server Receive Text Message 194
Server Receive Binary Message 195
Server Read Headers from Client 196
TsgcWebSocketHTTPServer 197
HTTP Server Requests 199
HTTP Dispatch Files 200
HTTP/2 Server 201
HTTP/2 Server Push 202

HTTP/2 Alternate Service 204
HTTP/2 Server Threads 205
HTTP 404 Error without Response Body 207
HTTP Server Sessions 208
HTTP Server Stream Video 210
Server SSL SChannel 211
TsgcWebSocketServer_HTTPAPI 214
HTTPAPI URL Reservation 216
HTTPAPI FineTune 218
HTTPAPI Server SSL 220
Self-Signed Certificates 221
HTTPAPI Disable HTTP/2 222
HTTPAPI Custom Headers 223
HTTPAPI Send Text Response 224
HTTPAPI Send File Response 225
HTTPAPI OnDisconnect not fired 227
TsgcWebSocketClient_WinHTTP 228
TsgcWebSocketFirewall 229
Firewall: Blacklist and Whitelist 236
Firewall: Brute Force Protection 238
Firewall: SQL Injection and XSS Detection 240
Firewall: Rate Limiting and Flood Protection 243
TsgcWSRateLimiter 246
TsgcWSCircuitBreaker 258
TsgcWSAPIKeyManager 269
TsgcWebSocketLoadBalancerServer 281
TsgcWebSocketProxyServer 282
TsgcIWWebSocketClient 283
TsgcWSConnection 285
Protocols 287
Protocols Javascript 289

Protocol MQTT 292
TsgcWSPClient_MQTT 294
Client MQTT Connect 295
Connect Mosquitto MQTT Servers 296
Client MQTT Sessions 297
Client MQTT Version 298
MQTT Publish Subscribe 299
MQTT Topics 300
MQTT Subscribe 301
MQTT Publish Message 302
MQTT Receive Messages 303
MQTT Publish and Wait Response 304
MQTT Clear Retained Messages 305
Protocol AMQP 306
TsgcWSPClient_AMQP 307
Client AMQP Connect 310
Client AMQP Disconnect 311
AMQP Channels 312
AMQP Exchanges 314
AMQP Queues 316
AMQP Publish Messages 319
AMQP Consume Messages 320
AMQP Get Messages 322
AMQP QoS 323
AMQP Transactions 324
Protocol AMQP1 326
TsgcWSPClient_AMQP1 328
Client AMQP1 Connect 331
Client AMQP1 Disconnect 332
Client AMQP1 Idle Timeout Connection 333
Client AMQP1 Connection State 334

Client AMQP1 Authentication 335
Client AMQP1 Azure MessageBus 336
AMQP1 Sessions 338
AMQP1 Links 340
AMQP1 Sender Links 341
AMQP1 Receiver Links 343
AMQP1 Send Message 345
AMQP1 Read Message 347
Protocol STOMP 348
TsgcWSPClient_STOMP 349
TsgcWSPClient_STOMP_RabbitMQ 350
TsgcWSPClient_STOMP_ActiveMQ 351
Protocol AppRTC 352
TsgcWSPServer_AppRTC 353
Protocol WebRTC 354
TsgcWSPServer_WebRTC 355
Protocol WebRTC Javascript 356
Protocol WAMP 357
TsgcWSPServer_WAMP 358
TsgcWSPClient_WAMP 359
Protocol WAMP Javascript 360
Subscribers 363
Publishers 364
Simple RPC 365
RPC Progress Results 366
Protocol WAMP2 368
TsgcWSPClient_WAMP2 369
Protocol Default 372
TsgcWSPServer_sgc 374
TsgcWSPClient_sgc 375
TsgcIWWSPClient_sgc 376

Protocol Default Javascript 377
Protocol Dataset 381
TsgcWSPServer_Dataset 382
TsgcWSPClient_Dataset 383
TsgcIWWSPClient_Dataset 384
Protocol Dataset Javascript 385
Protocol Dataset Replicate Table 388
Protocol Dataset Notify Updates 389
Protocol Files 390
TsgcWSPServer_Files 391
TsgcWSPClient_Files 392
TsgcWSMessageFile 393
How Send Files To Server 394
How Send Files To Clients 395
How Send Big Files 396
Protocol Presence 397
TsgcWSPServer_Presence 398
TsgcWSPresenceMessage 399
TsgcWSPClient_Presence 400
Protocol Presence Javascript 401
Protocol E2EE 404
TsgcWSPServer_E2EE 406
TsgcWSPClient_E2EE 407
APIs 408
API Binance 410
Binance Connect WebSocket API 418
Binance Subscribe WebSocket Channel 419
Binance Get Market Data 420
Binance Private REST API 421
Binance Trade Spot 422
Binance Private Requests Time 424

Binance Withdraw 425
API Binance Futures 426
API Binance Futures Trade 432
API SocketIO 433
API Coinbase 435
Coinbase Connect WebSocket API 440
Coinbase Subscribe WebSocket Channel 441
Coinbase Get Market Data 442
Coinbase Private REST API 443
Coinbase Private Requests Time 444
Coinbase Place Orders 445
Coinbase SandBox Account 446
API SignalRCore 447
API SignalR 453
API Kraken 456
API Kraken WebSockets Public 458
API Kraken WebSockets Private 464
API Kraken REST Public 467
API Kraken REST Private 469
API Kraken Futures 472
API Kraken Futures WebSockets Public 474
API Kraken Futures WebSockets Private 481
API Kraken Futures REST Public 487
API Kraken Futures REST Private 489
API Pusher 494
API Bitmex 502
Bitmex Connect WebSocket API 506
Bitmex Subscribe WebSocket Channel 507
How to Place a Bitmex Order 508
API Bitfinex 510
API Kucoin 514

Kucoin Connect WebSocket API 521
Kucoin Subscribe WebSocket Channel 522
Kucoin Get Market Data 523
Kucoin Private REST API 524
Kucoin Trade Spot 525
Kucoin Private Requests Time 527
API Kucoin Futures 528
Kucoin Futures Connect WebSocket API 533
Kucoin Futures Subscribe WebSocket Channel 534
Kucoin Futures Get Market Data 535
Kucoin Futures Private REST API 536
Kucoin Futures Trade 537
Kucoin Futures Private Requests Time 539
API 3Commas 540
API OKX 545
API XTB 551
API Bybit 554
API Forex 559
Forex Connect REST API 561
Forex Trade Orders 562
Forex Account Queries 564
Forex Market Data 565
Forex Connect WebSocket API 566
Forex Subscribe Streams 567
API Blockchain 569
API Cex 571
API Cex Plus 578
API Discord 582
API OpenAI 585
API MEXC 587
API MEXC Futures 591

API Bitget 595
API GateIO 597
API Deribit 599
API Crypto.com 602
API HTX 604
API Whatsapp 606
WhatsApp Create App 610
WhatsApp Phone Number Id 612
WhatsApp Token 613
WhatsApp Webhook 614
WhatsApp Security 615
WhatsApp Send Messages 616
WhatsApp Send Interactive Messages 619
WhatsApp Send Template Messages 623
WhatsApp Receive Messages and Status Notifications 625
WhatsApp Send Files 627
WhatsApp Download Media 629
API Telegram 630
Send Telegram Message With Inline Buttons 639
Send Telegram Message With Buttons 640
Send Telegram Message Bold 641
Telegram Chat not found as Bot 642
Telegram Sponsored Messages 643
Send Telegram Invoice Message 644
Telegram Get SuperGroup Members 645
Add Telegram Proxy 646
Register Telegram User 647
RCON 648
CryptoHopper 649
RTCMultiConnection 654
WebPush 656

TsgcWSAPIServer_WebPush 657
TsgcWebPush_Client 658
Extensions 659
PerMessage-Deflate 660
Deflate-Frame 661
MCP 662
TsgcWSAPIServer_MCP 663
MCP Server Sessions 665
MCP Server Tools 666
MCP Server Prompts 669
MCP Server Resources 672
MCP Server Roots 674
MCP Server Sampling 675
MCP Server Elicitation 676
TsgcWSAPIClient_MCP 678
MCP Client Tools 679
MCP Client Prompts 681
MCP Client Resources 683
MCP Client Roots 685
MCP Client Sampling 686
MCP Client Elicitation 687
OpenAI 688
TsgcHTTP_API_OpenAI 695
OpenAI Completion 696
OpenAI Chat 697
OpenAI Edit 698
OpenAI Audio 699
OpenAI Moderation 700
OpenAI RealTime 701
OpenAI Responses 702
OpenAI Speech 703

OpenAI Fine-Tuning 704
OpenAI Batch 705
OpenAI Uploads 706
OpenAI Applications 707
TsgcAIOpenAIAssistant 708
TsgcAIOpenAIAssistant File Search 710
TsgcAIOpenAIAssistant Streaming 712
TsgcAIOpenAIAssistant Function Calling 714
OpenAI Audio 716
TsgcAudioRecorderMCI 717
TsgcAudioPlayerMCI 718
TsgcTextToSpeechSystem 719
TsgcTextToSpeechGoogle 720
TsgcTextToSpeechAmazon 721
TsgcAIChat 722
TsgcAIOpenAIChatBot 723
TsgcAIOpenAITranslator 724
TsgcAIOpenAIEmbeddings 725
TsgcAIDatabaseVectorFile 726
TsgcAIDatabaseVectorPinecone 727
Embeddings Create Vectors 728
Embeddings ChatBot 729
Pinecone 730
Anthropic 733
TsgcHTTP_API_Anthropic 737
Anthropic Messages 738
Anthropic Vision 740
Anthropic Tool Use 742
Anthropic Models 744
Anthropic Batches 745
Anthropic Extended Thinking 746

Anthropic Documents 748
Anthropic Prompt Caching 750
Anthropic Citations 752
Anthropic Web Search 753
Anthropic Structured Outputs 755
Anthropic Files 757
Anthropic MCP Connector 759
Gemini 761
TsgcHTTP_API_Gemini 763
DeepSeek 764
TsgcHTTP_API_DeepSeek 765
DeepSeek Messages 766
DeepSeek Vision 767
DeepSeek Models 768
Gemini 769
Gemini Messages 771
Gemini Vision 773
Gemini Models 774
Gemini Structured Outputs 775
Gemini Token Counting 776
Gemini Embeddings 777
Gemini Tool Use 778
Ollama 780
TsgcHTTP_API_Ollama 782
Ollama Messages 783
Ollama Models 784
Ollama Embeddings 785
Grok 786
TsgcHTTP_API_Grok 787
Grok Messages 788
Grok Vision 789

Grok Models ... 790
Mistral AI ... 791
TsgcHTTP_API_Mistral ... 793
Mistral Messages ... 794
Mistral Vision ... 795
Mistral Models ... 796
Mistral Embeddings ... 797
IoT ... 798
IoT Amazon MQTT Client ... 799
IoT Azure MQTT Client ... 806
HTTP ... 811
HTTP2 ... 812
TsgcHTTP2Client ... 813
Request HTTP/2 Method ... 815
HTTP/2 Server Push ... 816
HTTP/2 Download File ... 817
HTTP/2 Partial Responses ... 818
HTTP/2 Headers ... 819
Client Close Connection ... 820
Client Keep Connection Active ... 821
HTTP/2 Reason Disconnection ... 822
Client Pending Requests ... 823
Client Authentication ... 824
HTTP/2 and OAuth2 ... 825
TsgcHTTP2ConnectionClient ... 826
TsgcHTTP2RequestProperty ... 827
TsgcHTTP2ResponseProperty ... 828
Apple Push Notifications ... 829
Register your APP with APNs ... 830
Generate a Remote Notification APNs ... 831
Sending Notification Requests to APNs ... 832

Token-Based Connection to APNs 833
Certificate-Based Connection to APNs 834
HTTP1 836
OAuth2 840
TsgcHTTP_OAuth2_Client 841
OAuth2 Client for Web Applications 845
OAuth2 Client for Desktop Applications 846
TsgcHTTP_OAuth2_Client_Google 847
TsgcHTTP_OAuth2_Client_Microsoft 848
Authorization Code Grant (RFC 6749) 849
Authorization Code with PKCE (RFC 7636) 851
Client Credentials Grant (RFC 6749) 853
Resource Owner Password Credentials Grant (RFC 6749) 855
Device Authorization Grant (RFC 8628) 857
DPoP - Demonstrating Proof of Possession (RFC 9449) 859
TsgcHTTP_OAuth2_Server 863
OAuth2 Server Example 866
OAuth2 Customize Sign-In HTML 870
OAuth2 Server Endpoints 871
OAuth2 Register Apps 872
OAuth2 Recover Access Tokens 873
OAuth2 Server Authentication 874
OAuth2 None Authenticate URLs 875
Authorization Code Grant 876
Client Credentials Grant 878
Resource Owner Password Credentials Grant 880
Refresh Token Grant 882
Device Authorization Grant (RFC 8628) 884
Token Revocation (RFC 7009) 886
Token Introspection (RFC 7662) 888
DPoP Validation (RFC 9449) 890

TsgcHTTP_OAuth2_Server_Provider 892
OAuth2 Provider Azure AD 894
OAuth2 Provider Private Endpoints 895
OAuth2 Provider Authentication 896
OAuth2 Provider Requests 898
JWT 899
TsgcHTTP_JWT_Client 901
TsgcHTTP_JWT_Server 904
WebAuthn 905
TsgcWSAPIServer_WebAuthn 906
WebAuthn Registration 910
WebAuthn Registration Request 913
WebAuthn Registration Response 914
WebAuthn Registration Result 916
WebAuthn Authentication 917
WebAuthn Authentication Request 919
WebAuthn Authentication Response 920
WebAuthn Authentication Result 921
WebAuthn MDS 922
WebAuthn Authorization 923
WebAuthn Authorization HTTP 924
WebAuthn Authorization WebSocket 925
Webauthn Javascript Client 926
Amazon SQS 930
Google OAuth2 Keys 935
Google Service Accounts 941
Google Cloud Pub/Sub 945
Google Calendar 954
Google Calendar Sync Calendars 960
Google Calendar Sync Events 961
Google Calendar RefreshToken 962

Google Calendar Service Account 963
Google Cloud FCM 964
TsgcWebView2 967
WebView2 Navigation 969
WebView2 JavaScript 971
WebView2 Cookies 973
WebView2 Downloads 975
WebView2 Settings 977
WebView2 Advanced Features 979
TsgcUDPCLient 982
TsgcUDPServer 983
STUN 984
TsgcSTUNClient 985
STUN Client UDP Retransmissions 987
STUN Client Long Term Credentials 988
STUN Client Attributes 989
TsgcSTUNServer 990
STUN Server Long Term Credentials 991
STUN Server Alternate Server 992
TURN 993
TsgcTURNClient 994
TURN Client Allocate IP Address 996
TURN Client Create Permissions 997
TURN Client Send Indication 998
TURN Client Channels 999
TsgcTURNServer 1000
TURN Server Long Term Credentials 1001
TURN Server Allocations 1002
ICE 1003
TsgcICEClient 1004
ICE Gather Candidates 1005

ICE Pair Candidates 1006
TsgcRTCPeerConnection 1007
RTCPeerConnection WebSocket Server 1009
RTCPeerConnection WebSocket Client 1010
RTCPeerConnection STUN TURN 1011
RTCPeerConnection Signaling 1012
RTCPeerConnection ICE 1013
RTCPeerConnection DTLS 1014
RTCPeerConnection Data 1015
Datasnap 1016
TsgcWSHTTPWebBrokerBridgeServer 1017
TsgcWSHTTP2WebBrokerBridgeServer 1019
TsgcWSServer_HTTPAPI_WebBrokerBridge 1020

OpenAPI 1021

OpenAPI 1021
OpenAPI Parser Pascal 1022
OpenAPI Additional Properties 1027
OpenAPI Command Line 1029
OpenAPI Library 1031
OpenAPI API 1033
OpenAPI Client 1034
OpenAPI Amazon AWS 1037
OpenAPI Amazon AWS Credentials 1043
OpenAPI Amazon AWS S3 1045
OpenAPI Google Cloud 1046
OpenAPI Google Cloud OAuth2 1051
OpenAPI Google Cloud Service Accounts 1054
OpenAPI Google Cloud PubSub 1059
OpenAPI Google Cloud Calendar 1060
OpenAPI Google Drive 1061

OpenAPI Microsoft 1063
OpenAPI Microsoft Tenant 1068
OpenAPI Microsoft Register Application 1069
OpenAPI Microsoft OAuth2 Code 1072
OpenAPI Microsoft OAuth2 Credentials 1074
OpenAPI Microsoft Graph 1076
APIs 1077
AbstractApi Geolocation 1078

Demos 1079

Server Chat 1079
Client Chat 1081
Client 1082
Client MQTT 1083
Client SocketIO 1085
Server Monitor 1086
Server Snapshots 1089
Client Snapshots 1090
Upload File 1091
Server Authentication 1093
KendoUI_Grid 1094
ServerSentEvents 1096
Server WebRTC 1097
Server AppRTC 1098
Telegram Client 1100

Third-parties 1102

Coturn 1102

License 1104

License 1104

Index **1106**

PDF-Back-Cover **1114**

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.
- **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.

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_FXCM:** FXCM 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_AppRTC:** WebRTC Server based on AppRTC Google Project.
- **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.
- **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.
- **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.
- **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.

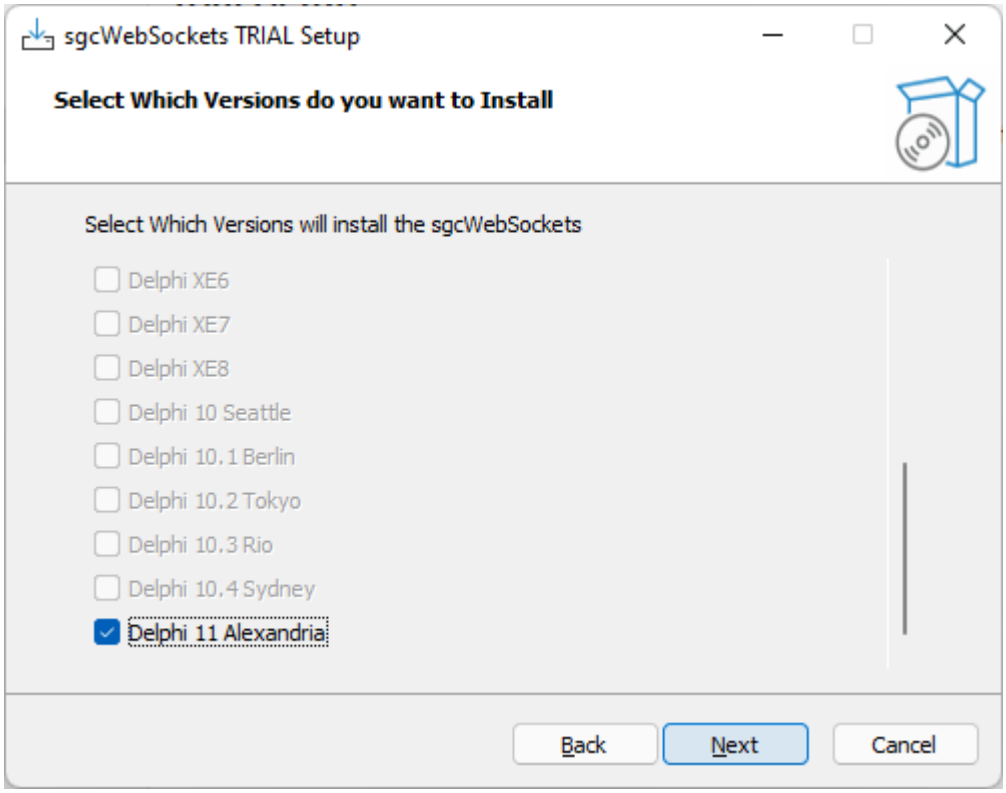
Install Setup

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

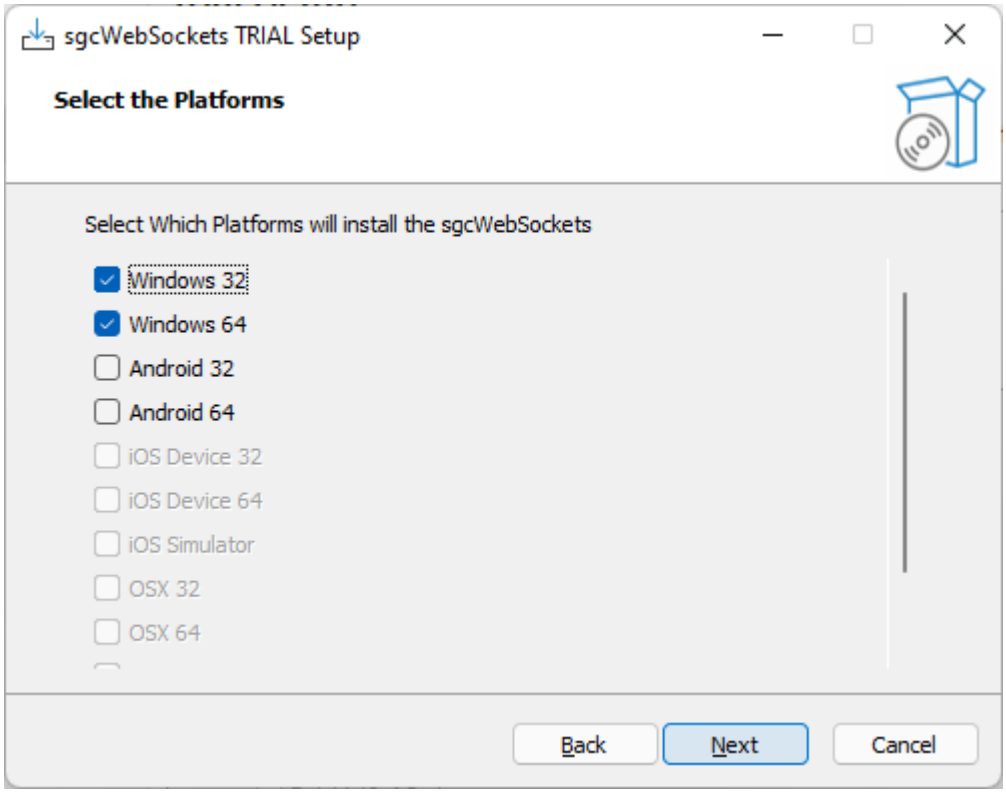
If you use the Windows Setup to install sgcWebSockets library, the installation is guided and very simple. If there is any error while installing, please refer to Install Errors page and you can try to install the package manually.

Trial Setup

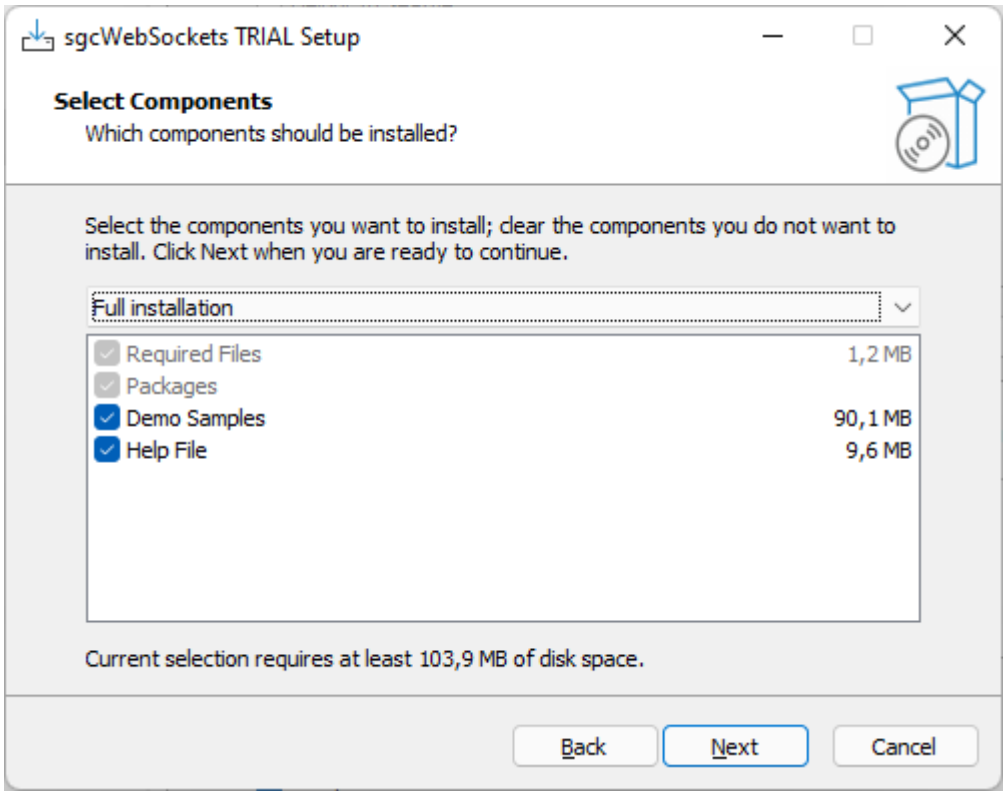
- Execute the Trial Installer.
- The Trial setup requires Admin privileges.
- The installer will show a list of Delphi / CBuilder / Rad Studio versions and by default the downloaded version will be enabled. If this version is NOT detected by the installer, the installer will extract the files but won't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

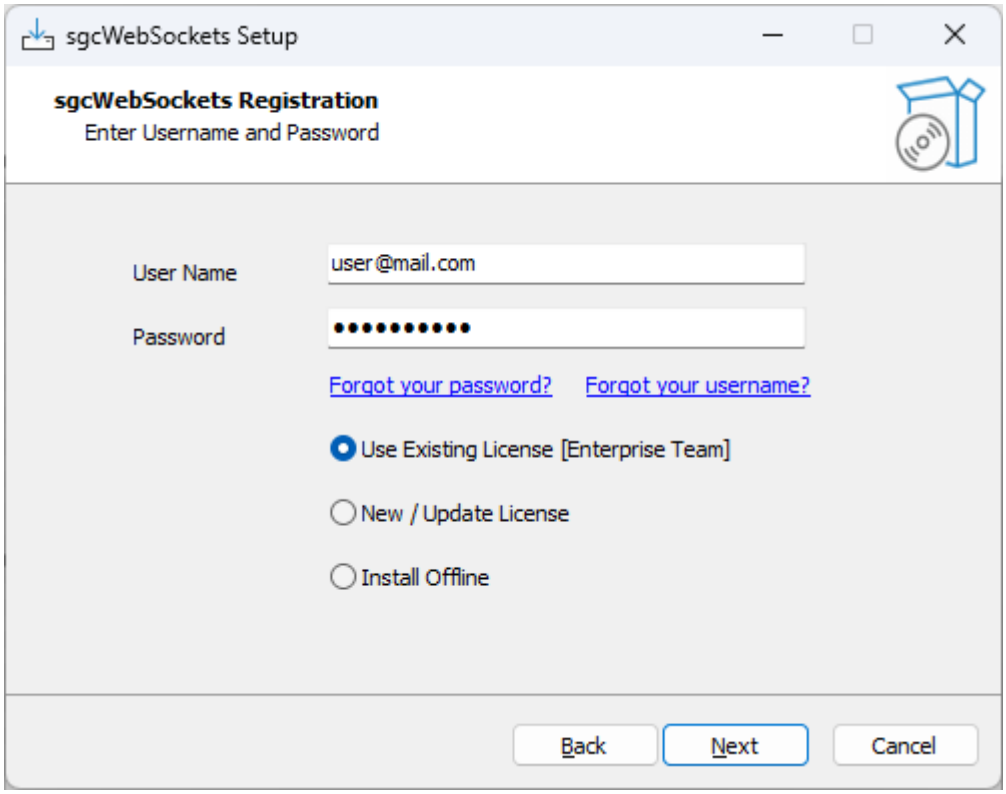


- Finally, it will extract the files, compile and install the package and register the required paths in the IDE.

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.

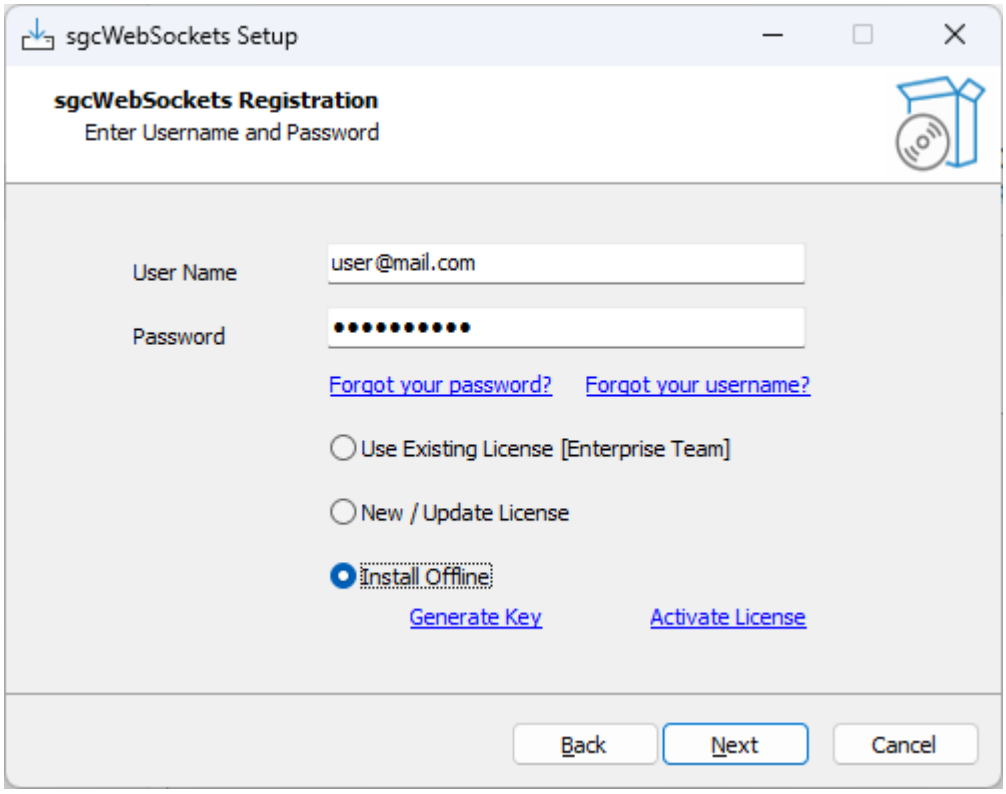
- Execute the Installer.
- The installer runs with the lowest privileges (if it runs as admin, it cannot be installed on network drives). If the destination requires admin privileges, run the setup as administrator.
- First you must set the username/password of your private eSeGeCe account. This only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



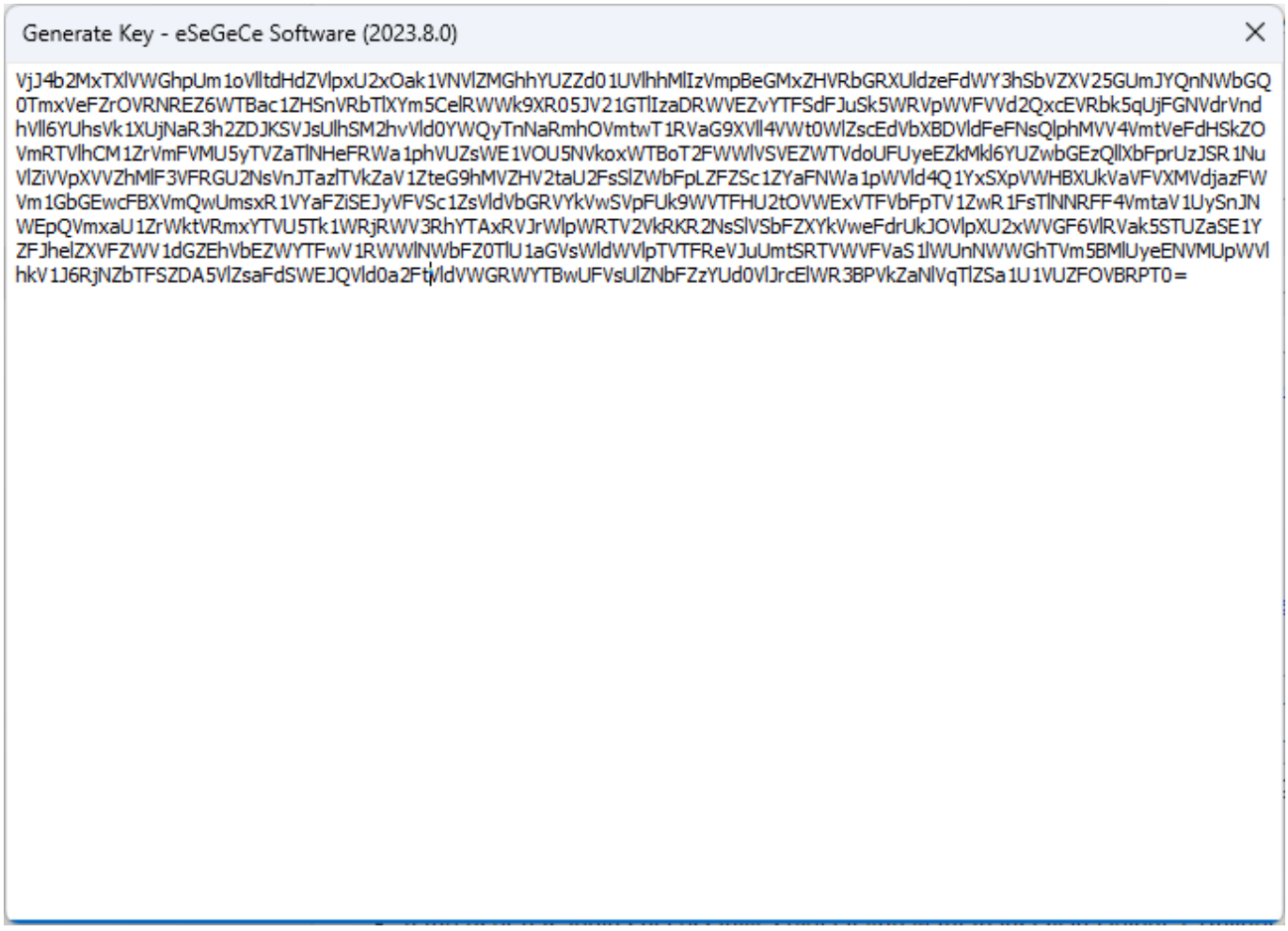
- There are 3 options:
  - **Use Existing License:** if this version has been already installed, the option will be selected by default. It will use the latest configuration for this version.
  - **New / Update License:** if this version has not been installed previously, this option will be selected 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

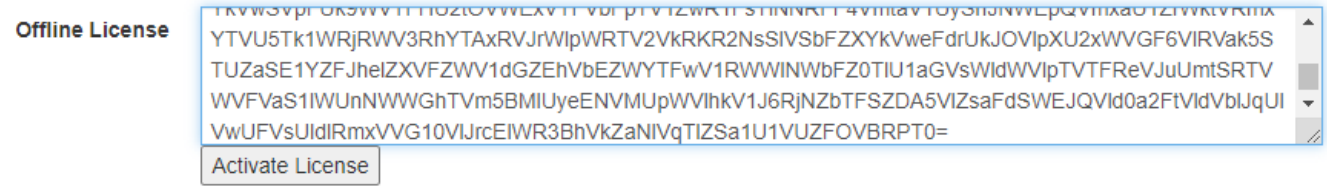
This option generates a key that will be used to activate the license.



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

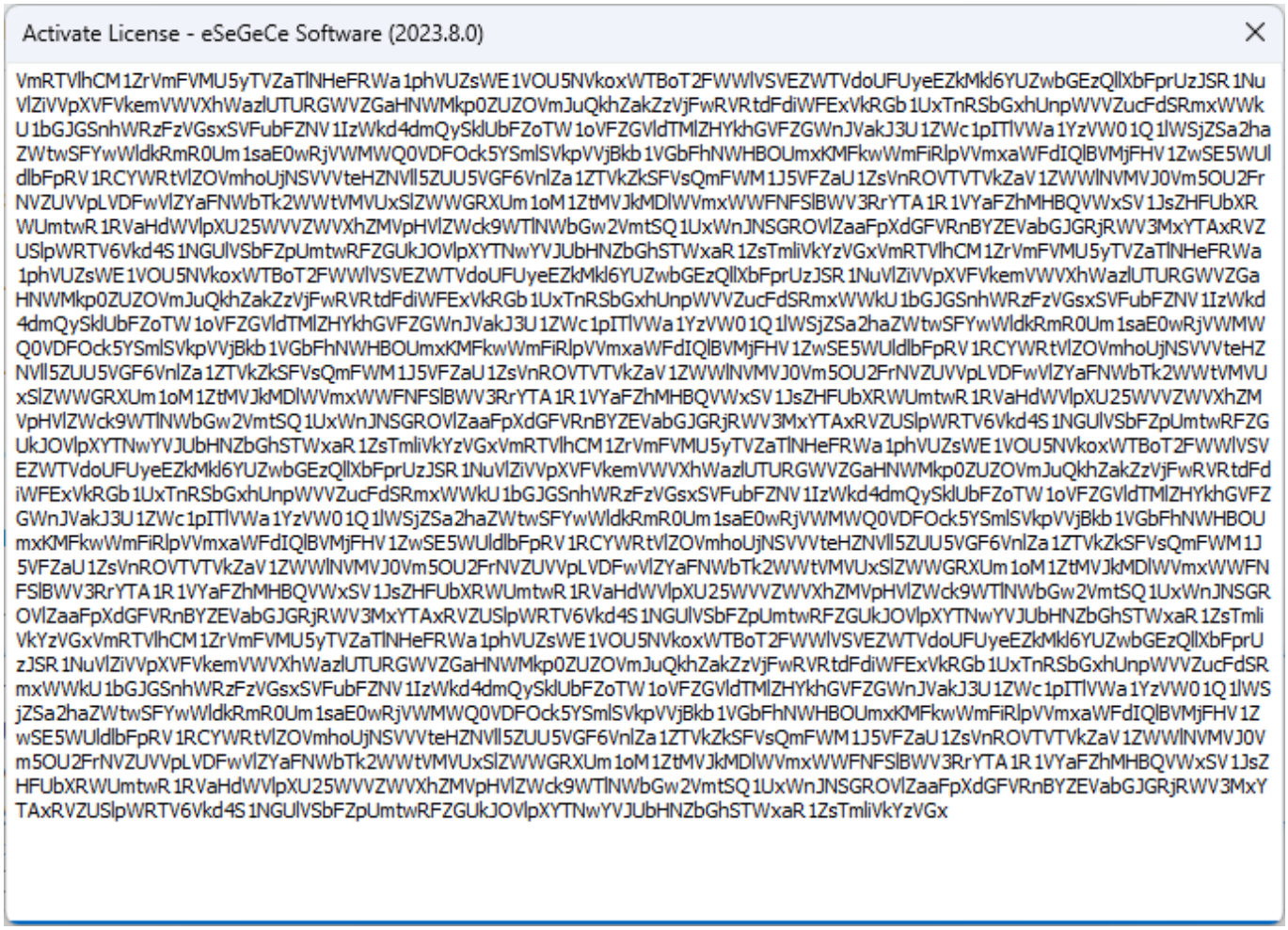


Select the subscription to activate and paste the key.

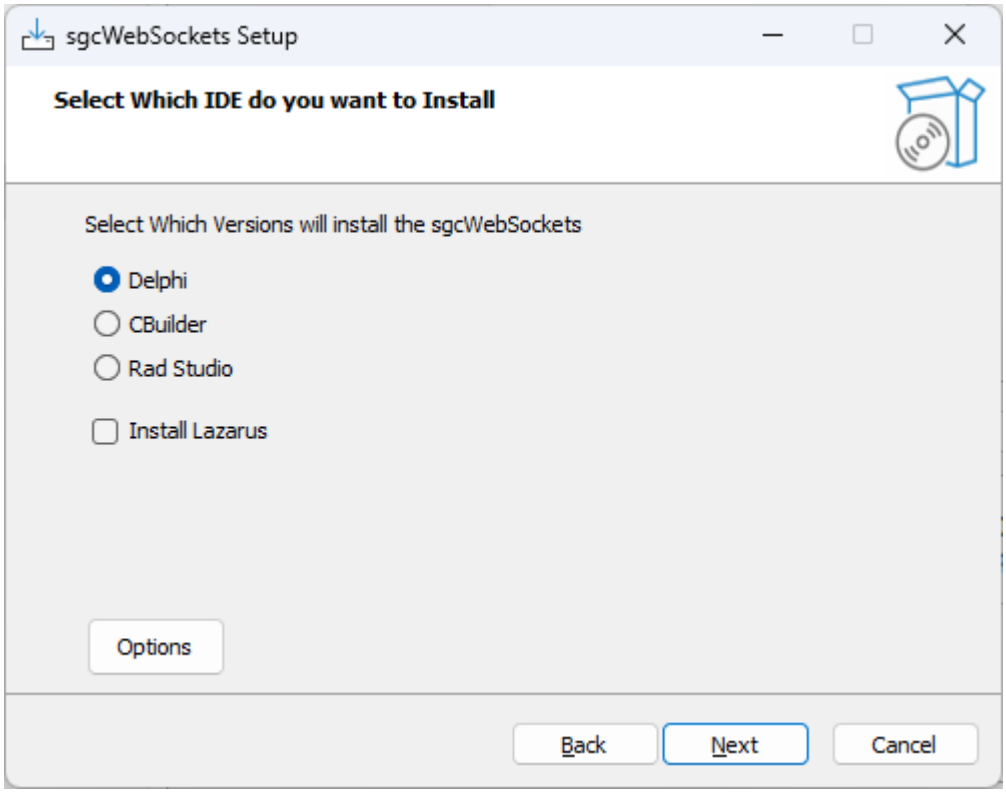


Activate License

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

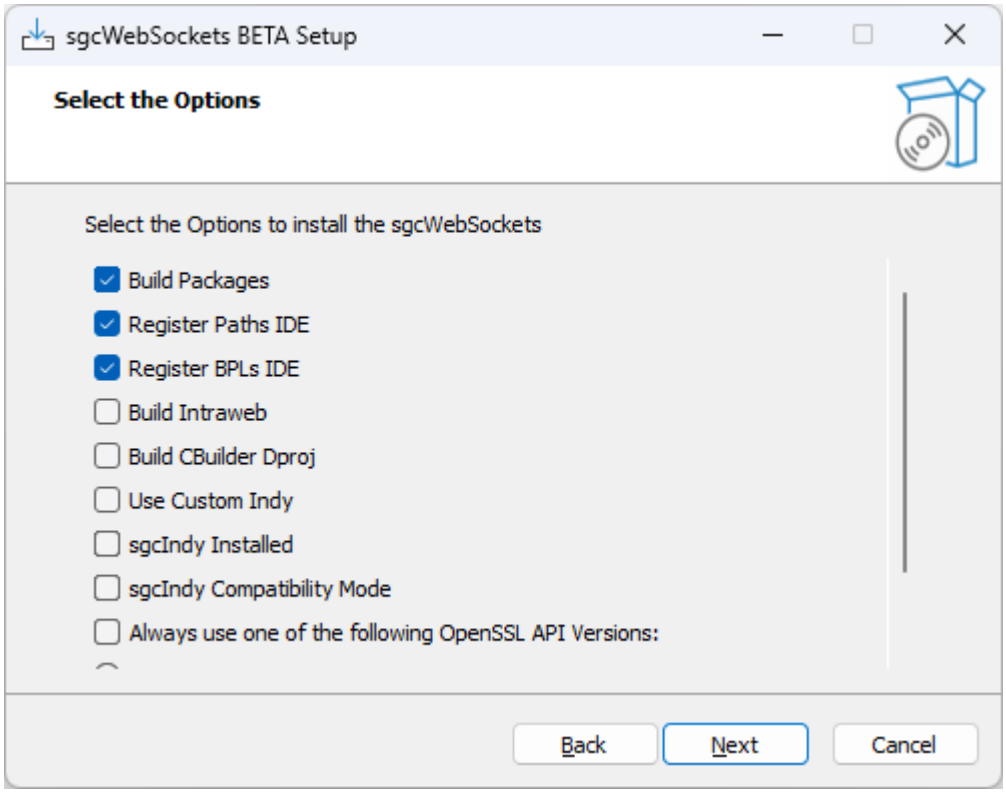


- If the license has been activated successfully, select if you want to install in Delphi, CBuilder or Rad Studio IDE. There is a check to extract the required lazarus files (Lazarus requires to install the package manually).

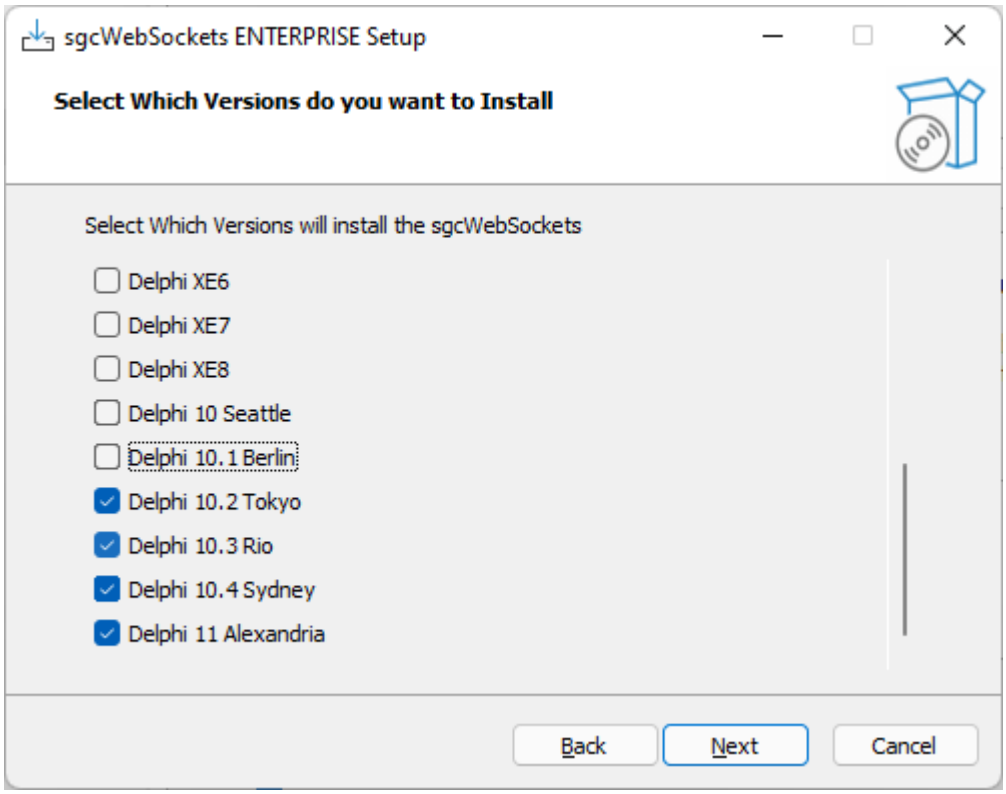


- There are some options that can be customized every time you use the installer, press the button **Options** to access these properties.
  - **Build Packages:** if selected, the installer will try to build the packages.
  - **Register Paths IDE:** if selected, the installer will register the required 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.

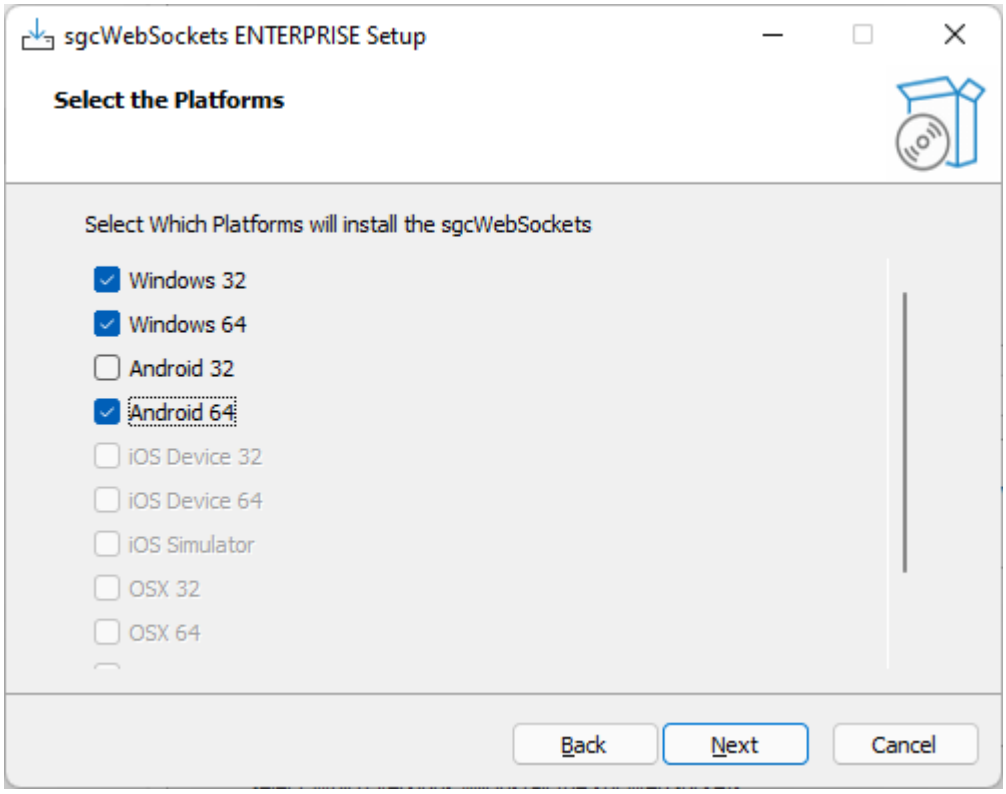
- The following options are only available for licenses with source code:
  - **Build Intraweb:** if selected, the installer will install the required Intraweb files (disabled by default). Currently installs the Intraweb XVI version.
  - **Build CBuilder Dproj:** if selected, the installer will build 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.



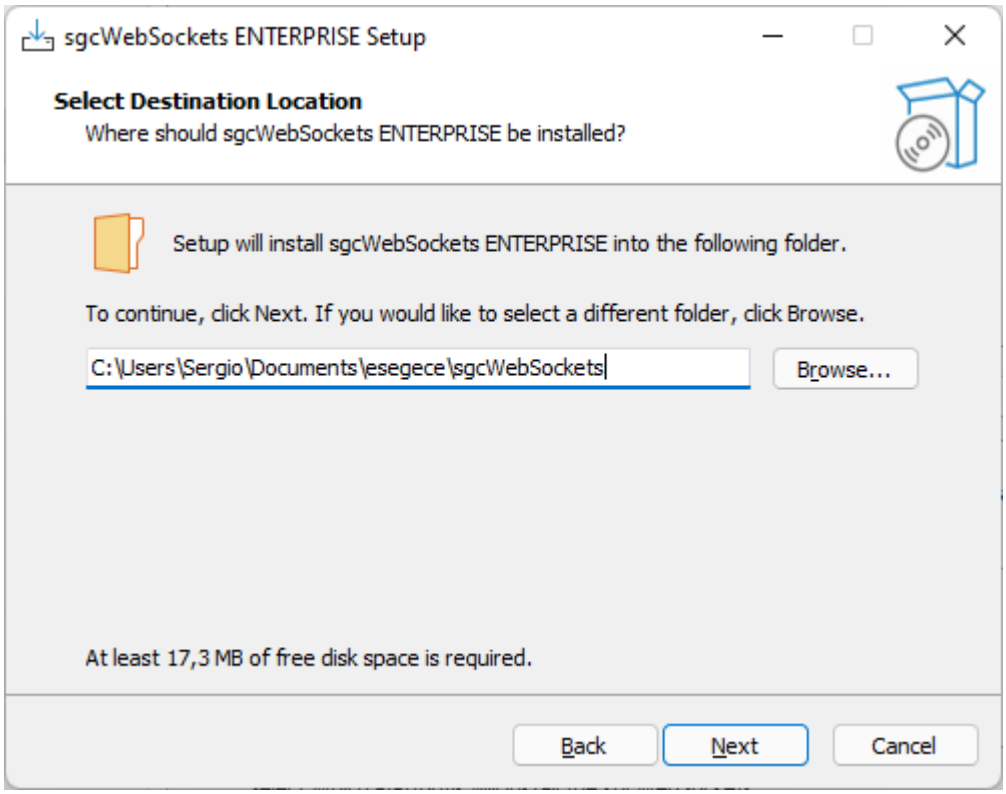
- Now you can select which IDE versions you want to install. Only those IDE versions that the installer detects as installed will be available.



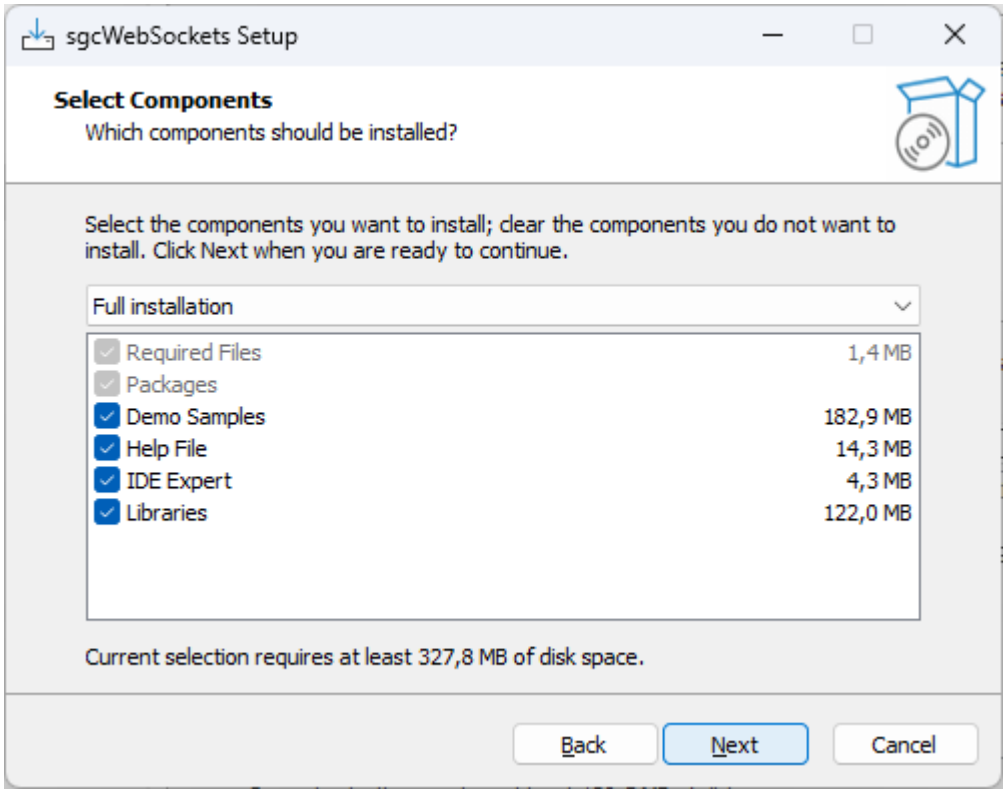
- The next step is to select the platforms.



- 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.

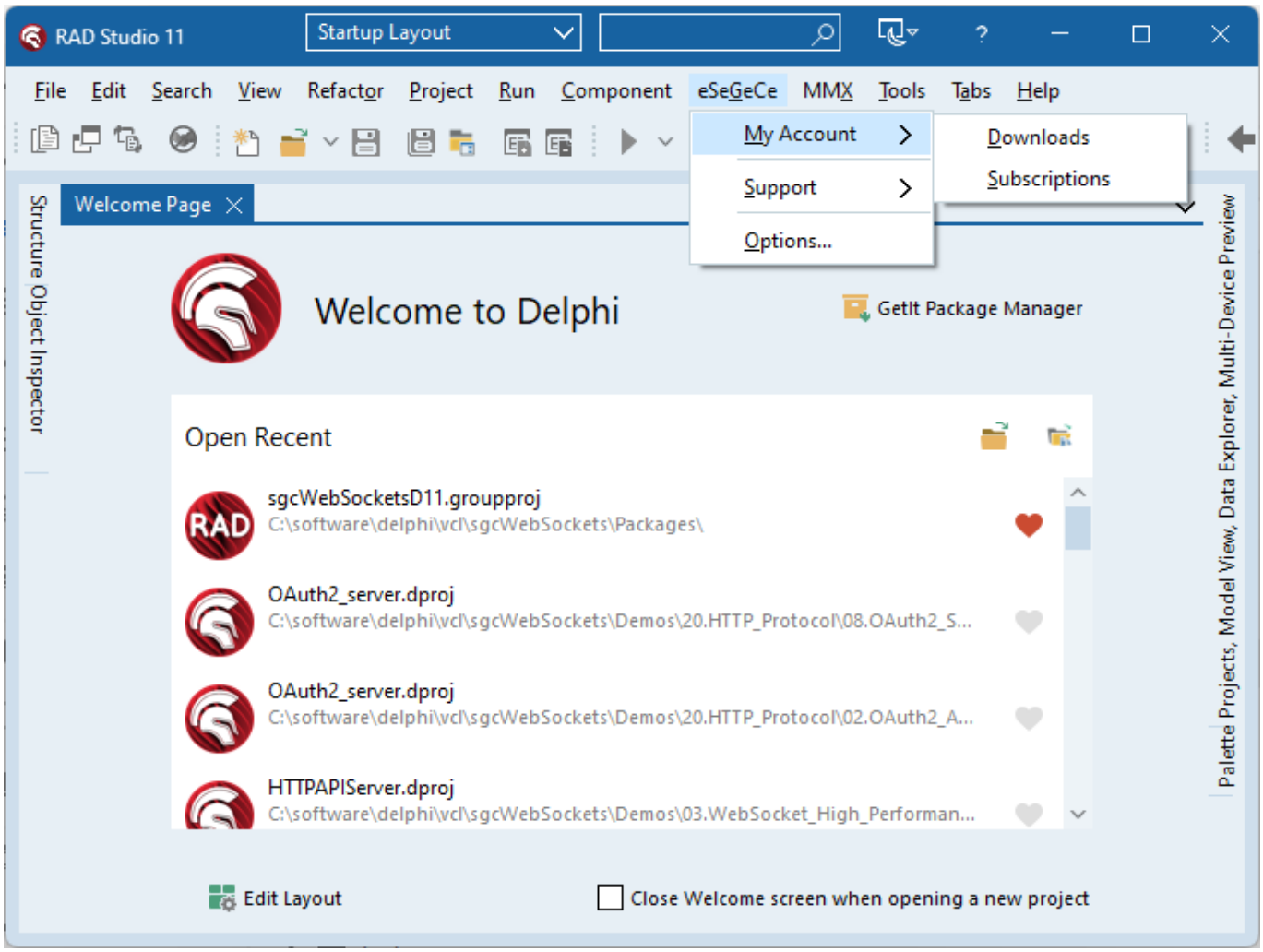
```
/platforms=Win32-Win64
```

/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:

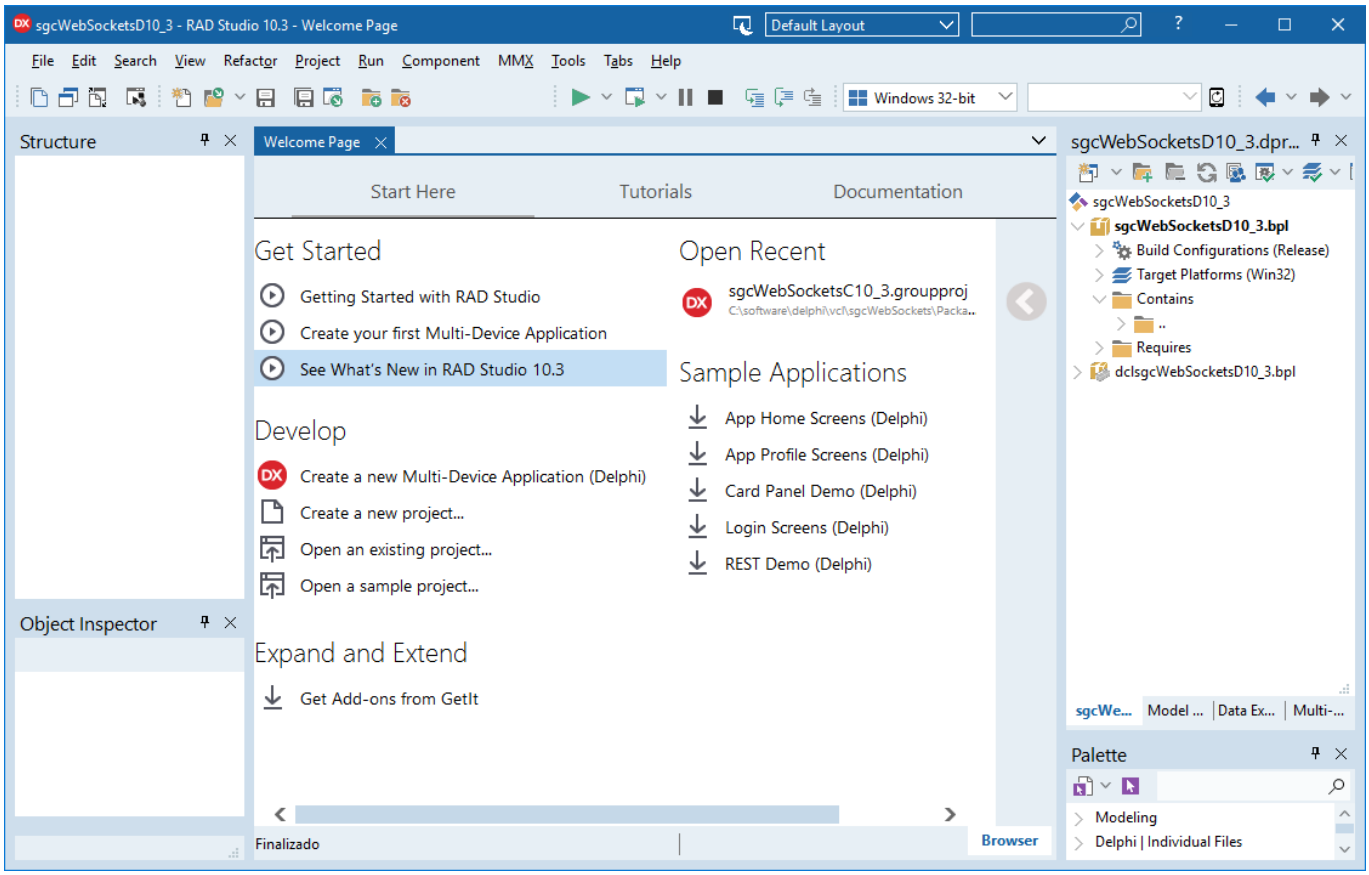


- **MyAccount:** direct access to the Downloads menu (where you can download the latest version of beta) and to the Subscriptions, to manage your license or renew an expired license.
- **Support:** direct access to HelpDesk or Forum with automatic login. Documentation and Contact Us form is available 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

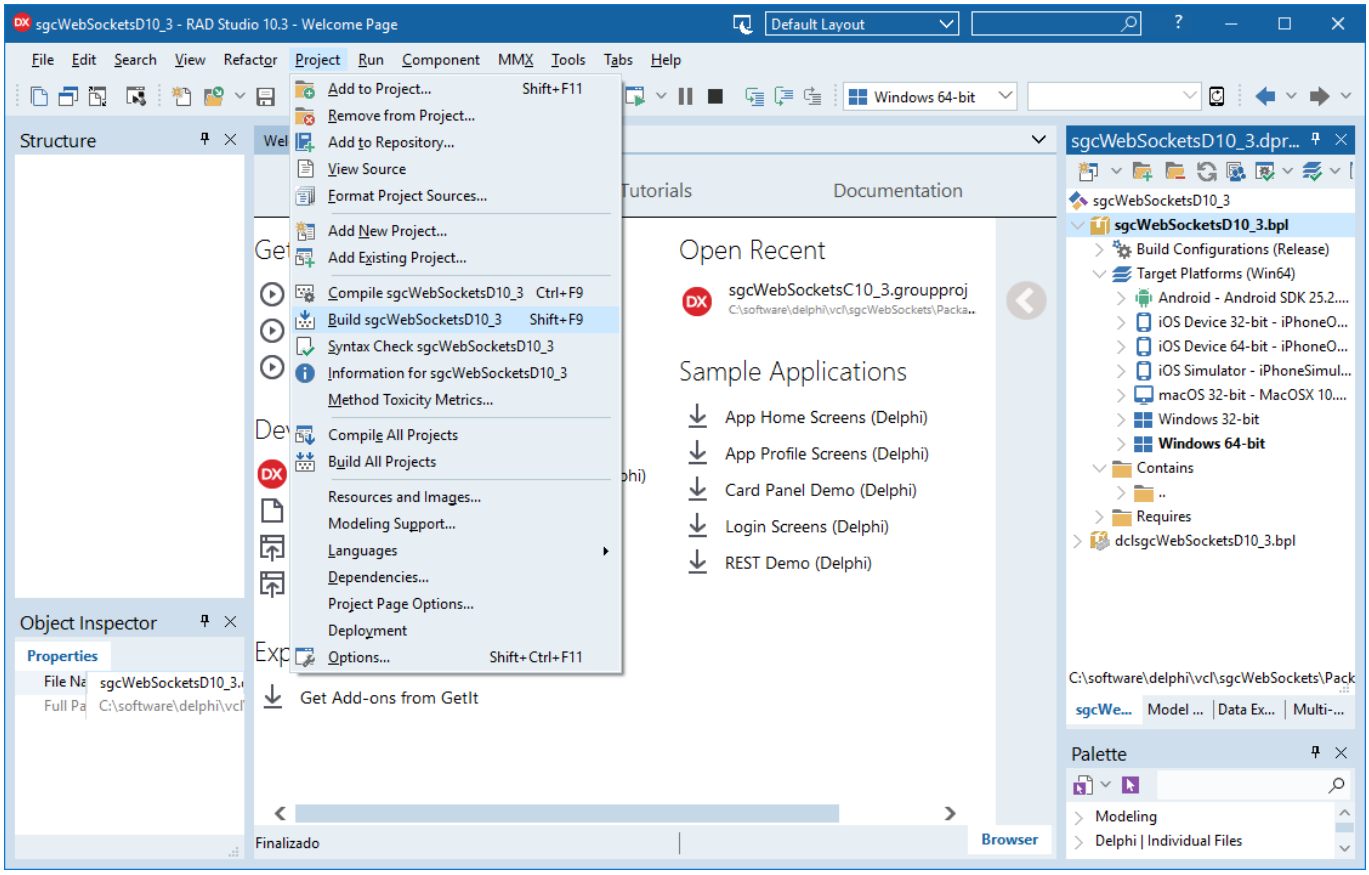
Follow next steps to install sgcWebSockets package, screenshots use Delphi 10.3 version.

1. Open sgcWebSocketsD10_3 group project.

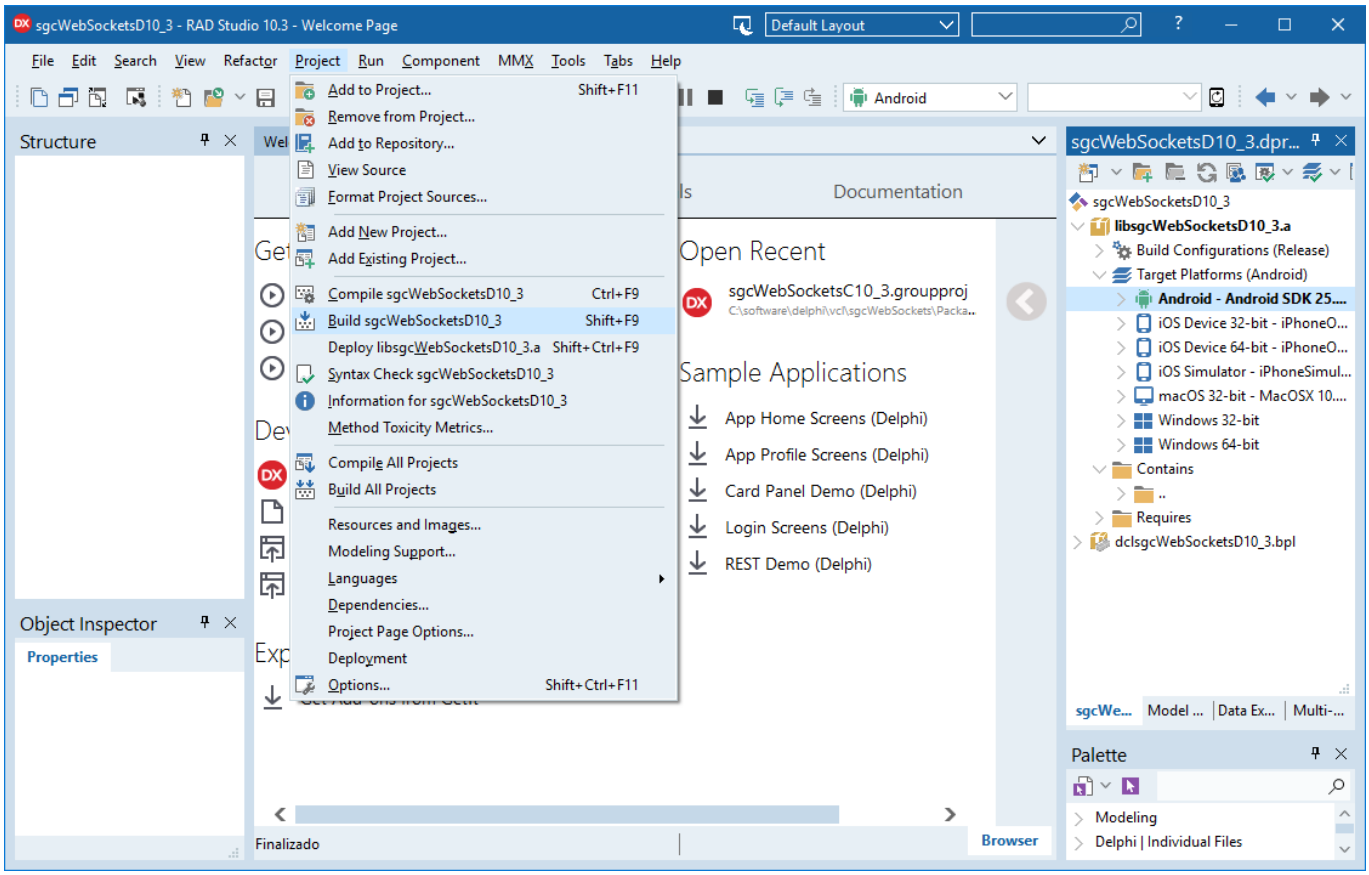


2. Now we must **compile first runtime packages** (name starts with sgcWebSockets). There is one package for every target platform and this depends on the Delphi version, so **select target platform one by one and build** every package.

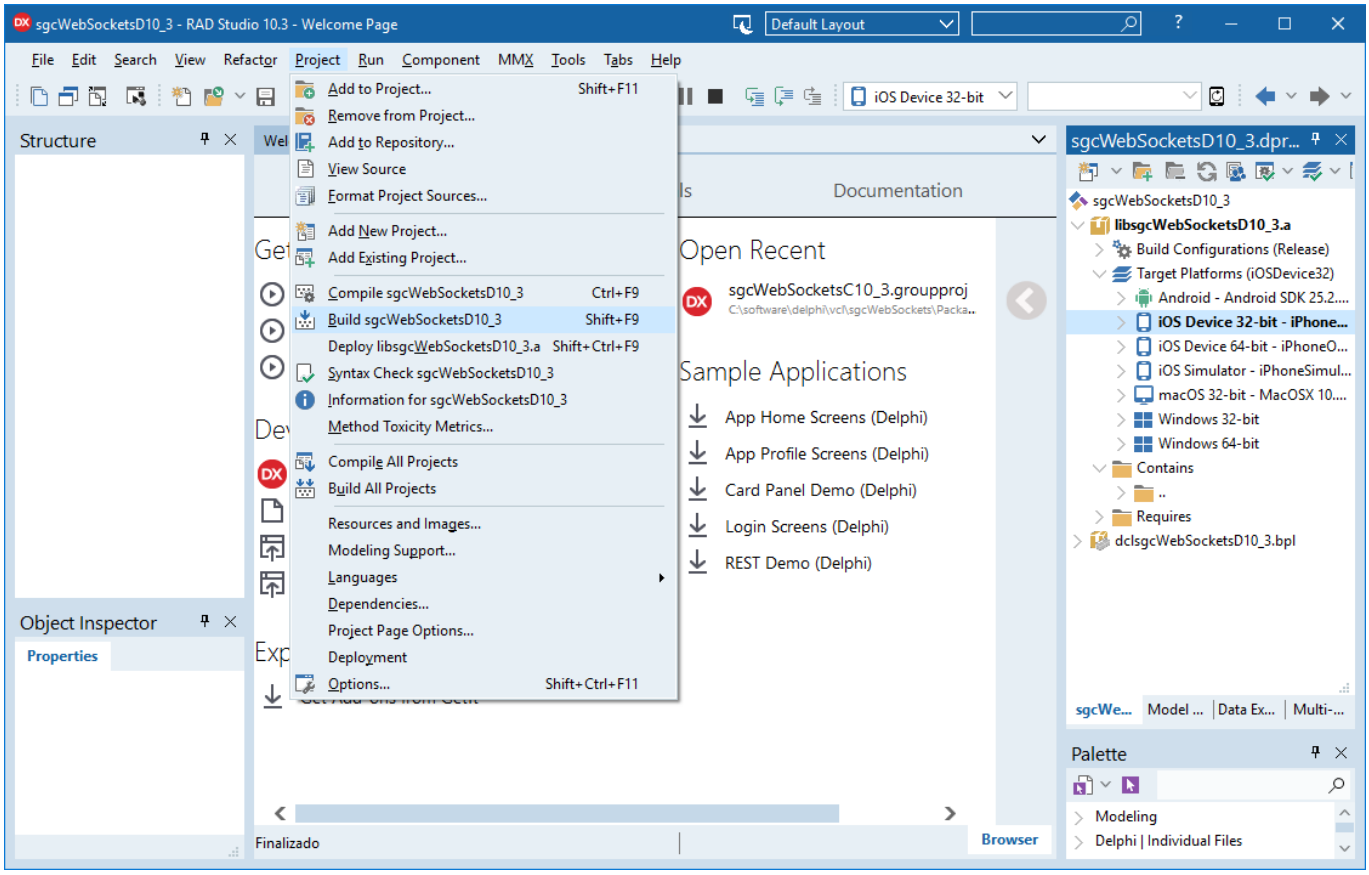
3. Select win64 as Target platform and build package.



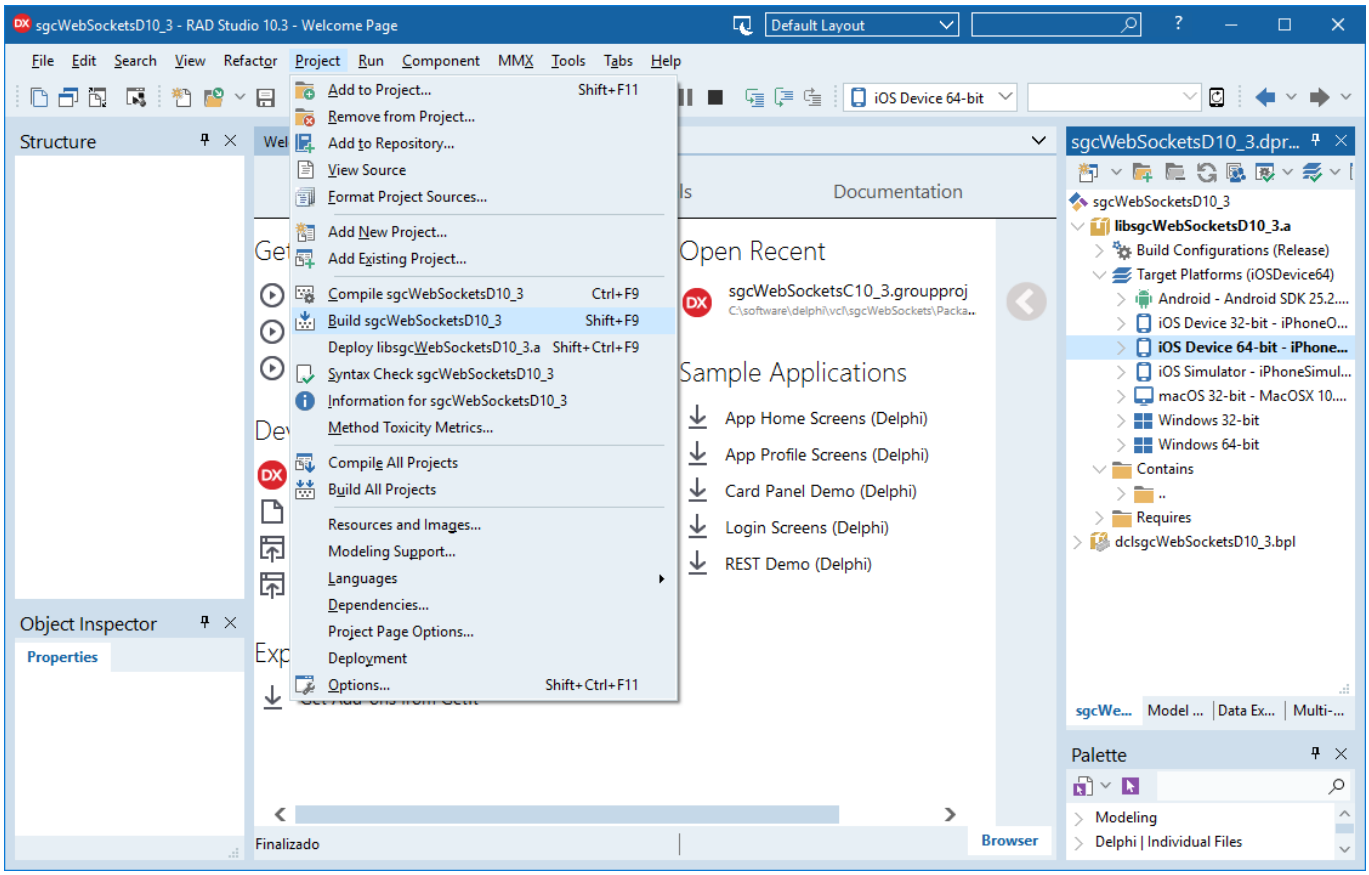
4. Select Android as Target Platform and build package.



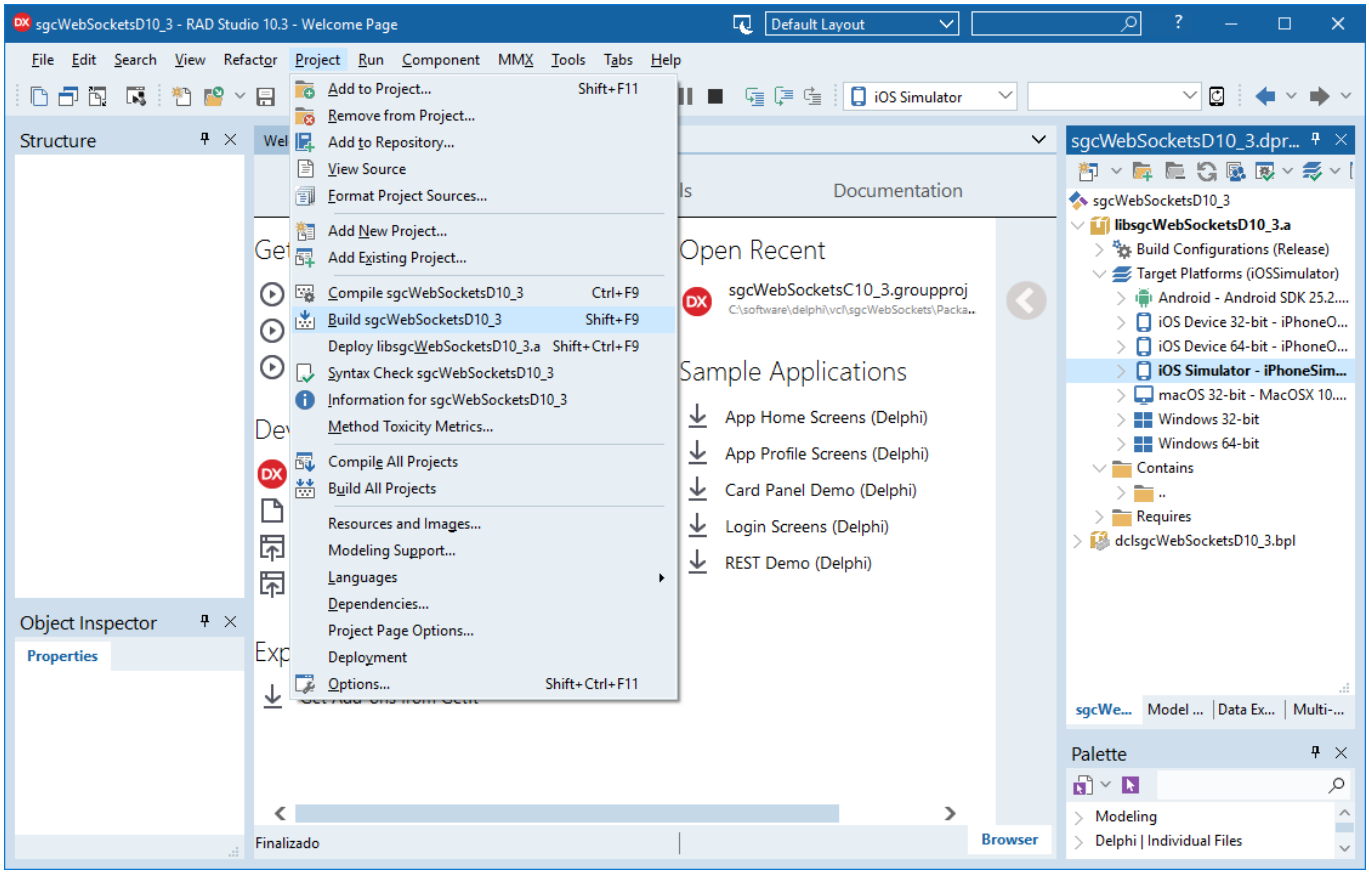
5. Select **iOS Device 32** as Target Platform and build package.



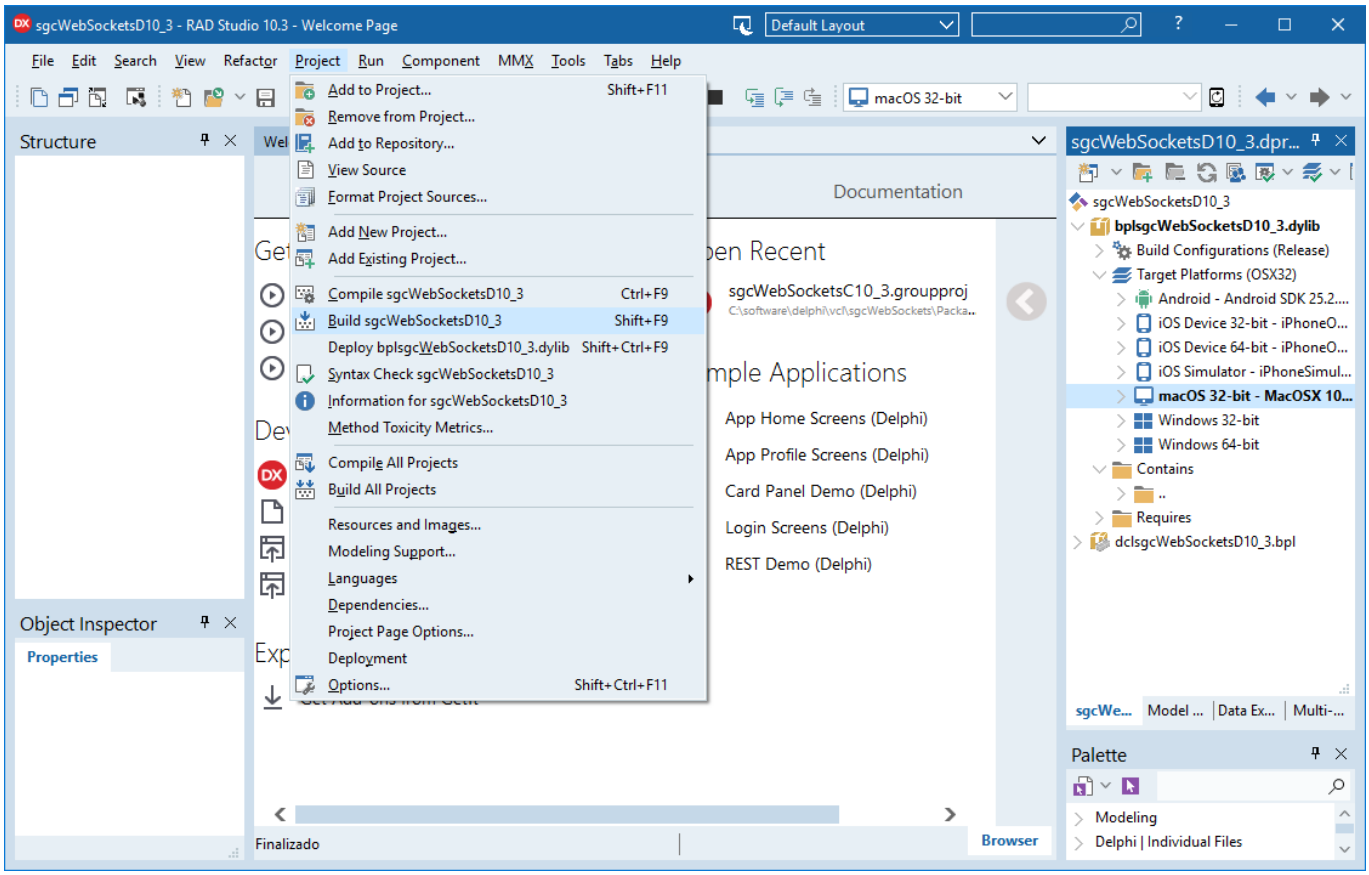
6. Select **iOS Device 64** as Target Platform and build package.



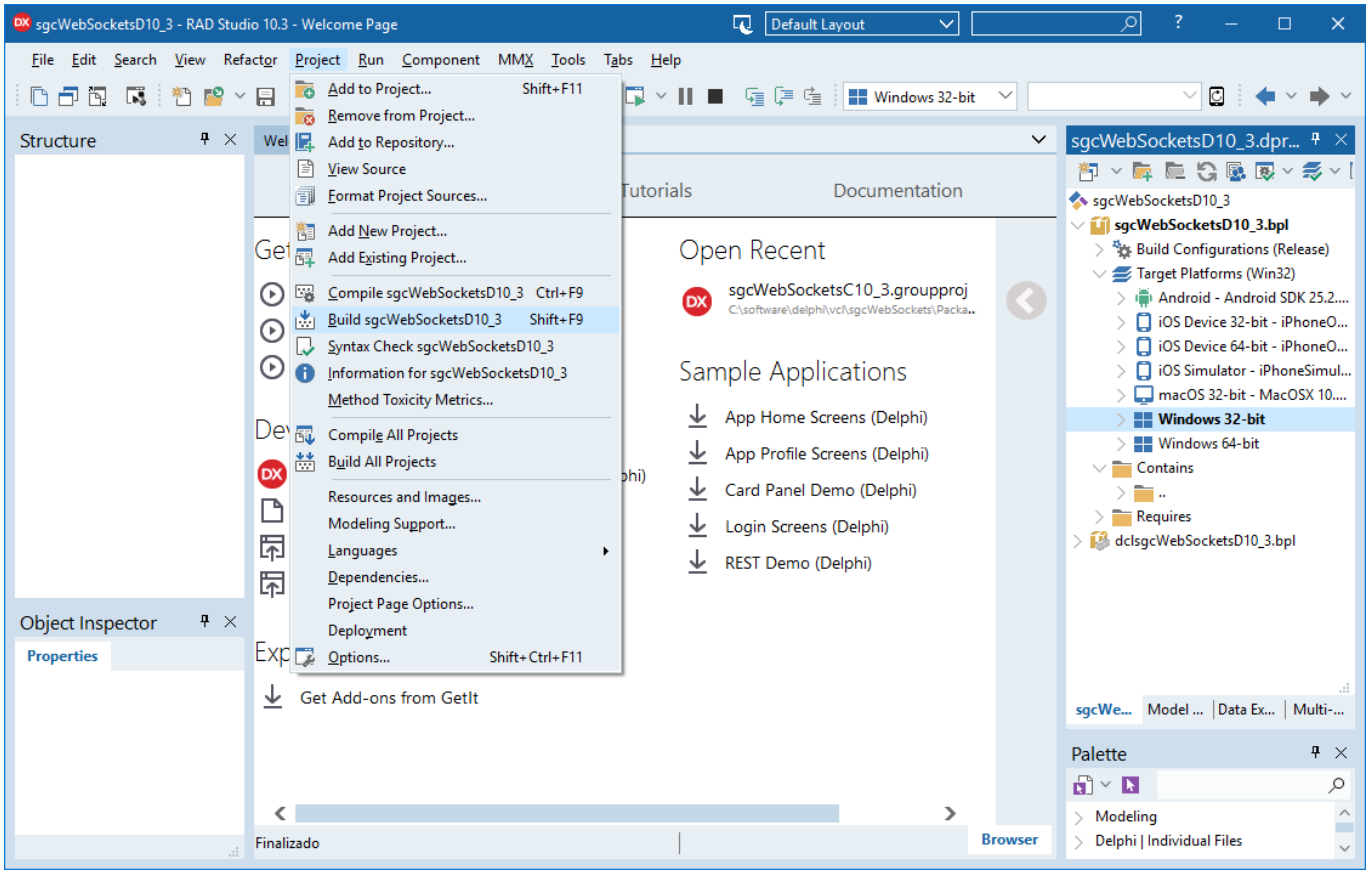
7. Select **iOS Device** Simulator as Target Platform and build package.



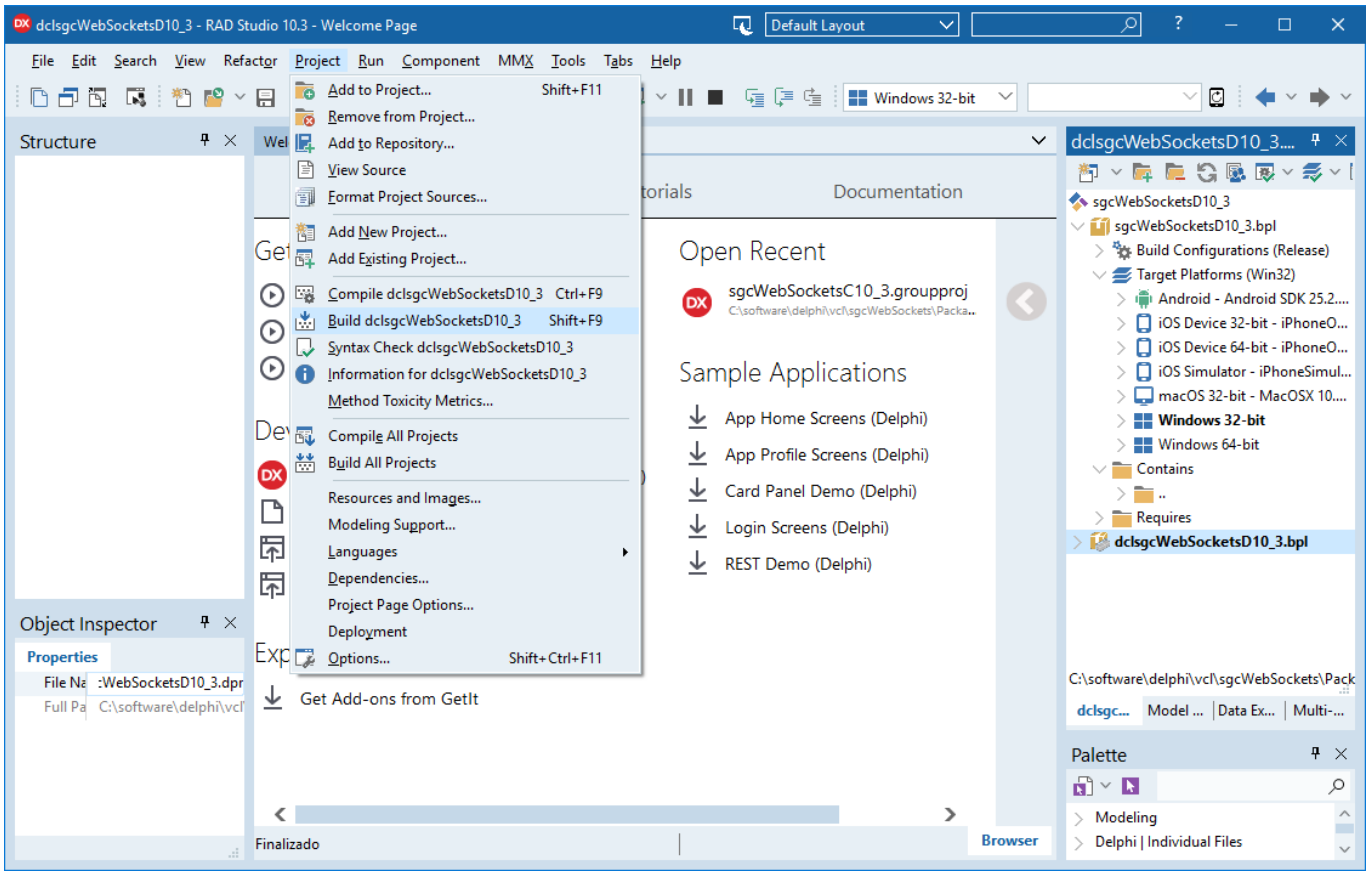
8. Select **MacOS 32** as Target Platform and build package.

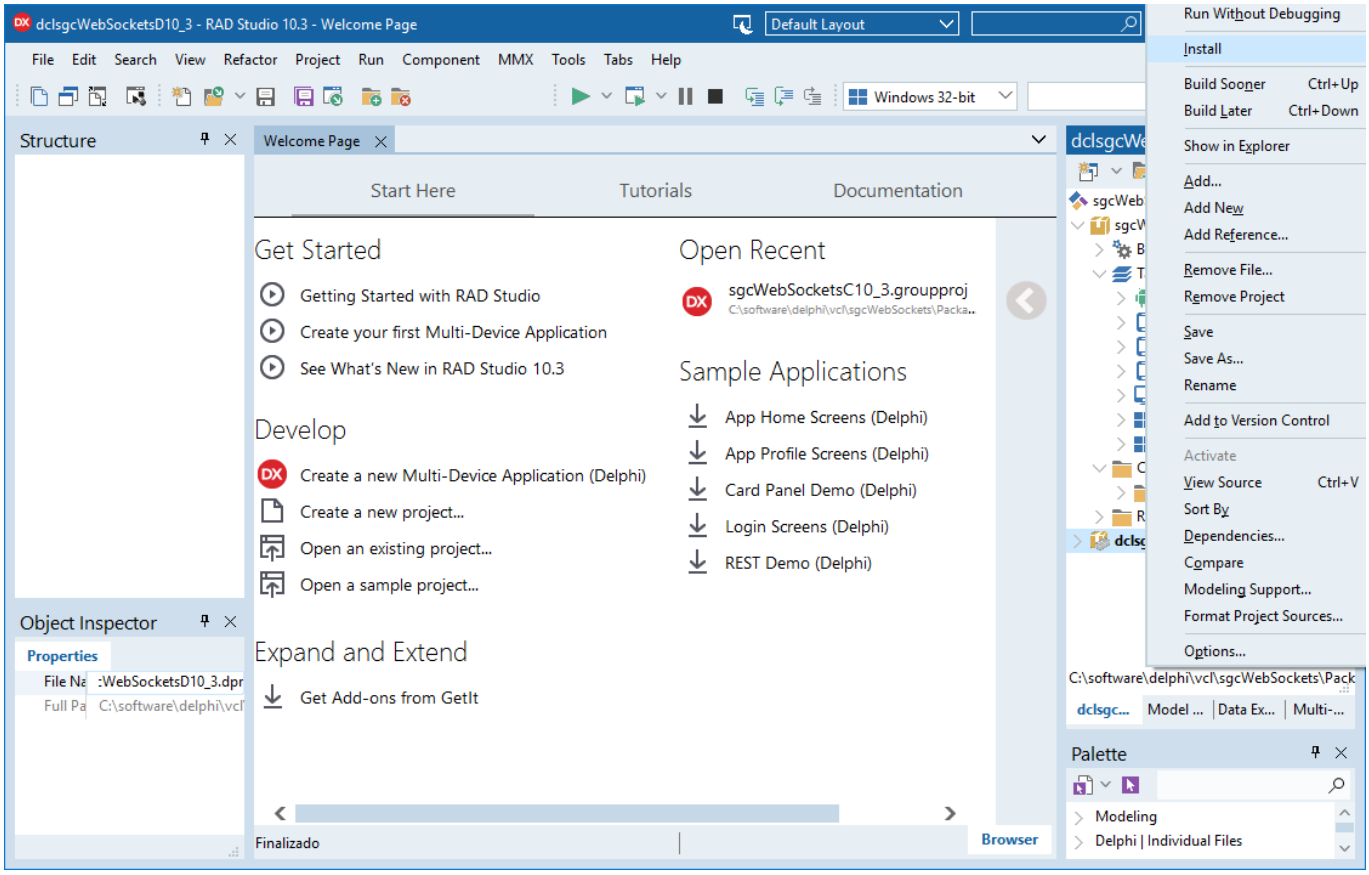


9. Select **Win32** as Target Platform and build package.

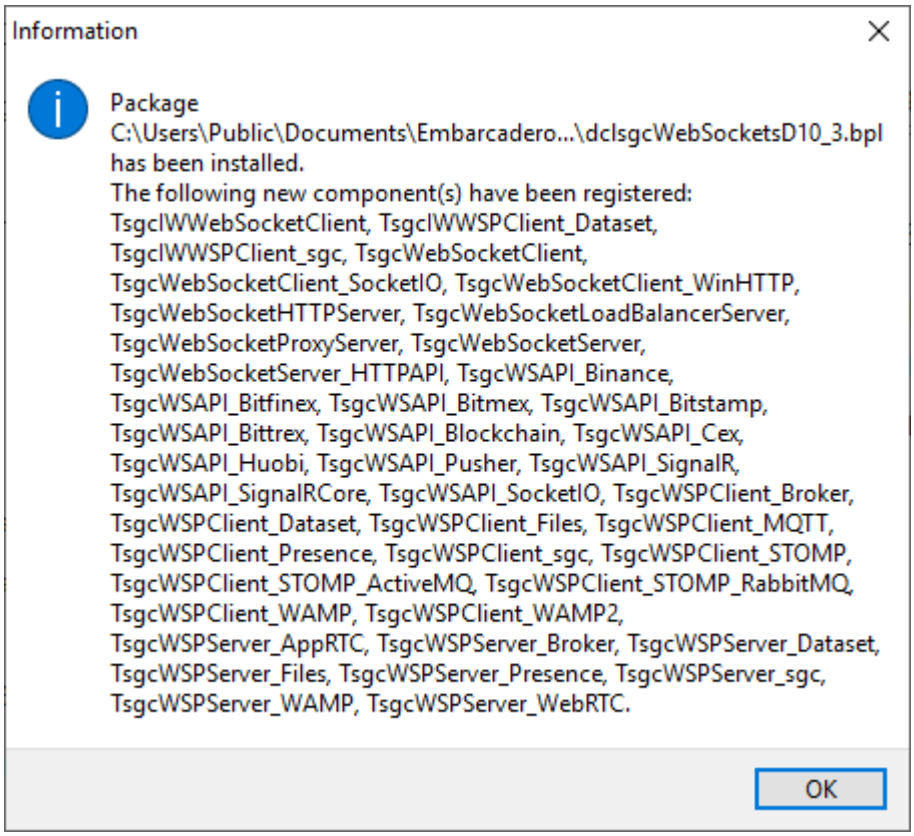


10. Once all runtime packages are compiled, select **design time package** (name starts with dcl) and first **build** and then **install** (design time packages only have Win32 as target platform).

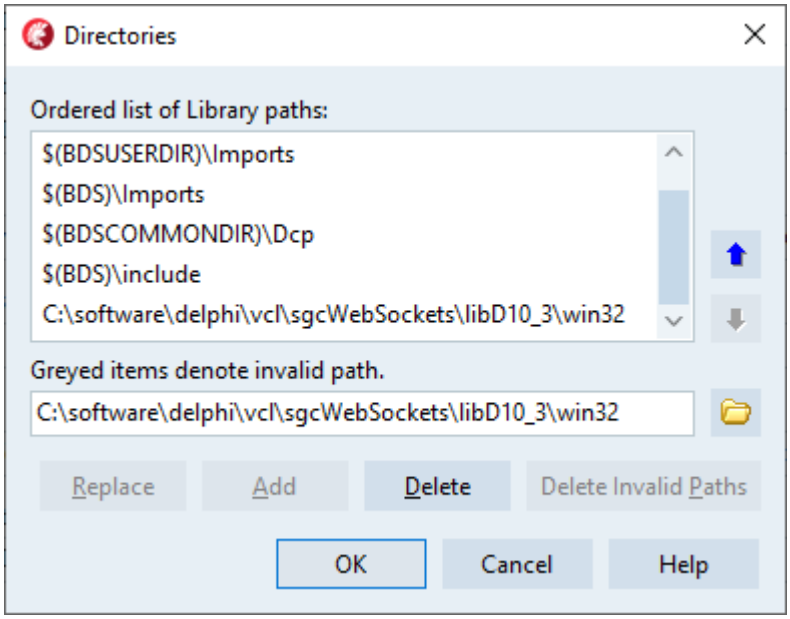




11. If installation is **successful** you will see a message with all components installed.



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

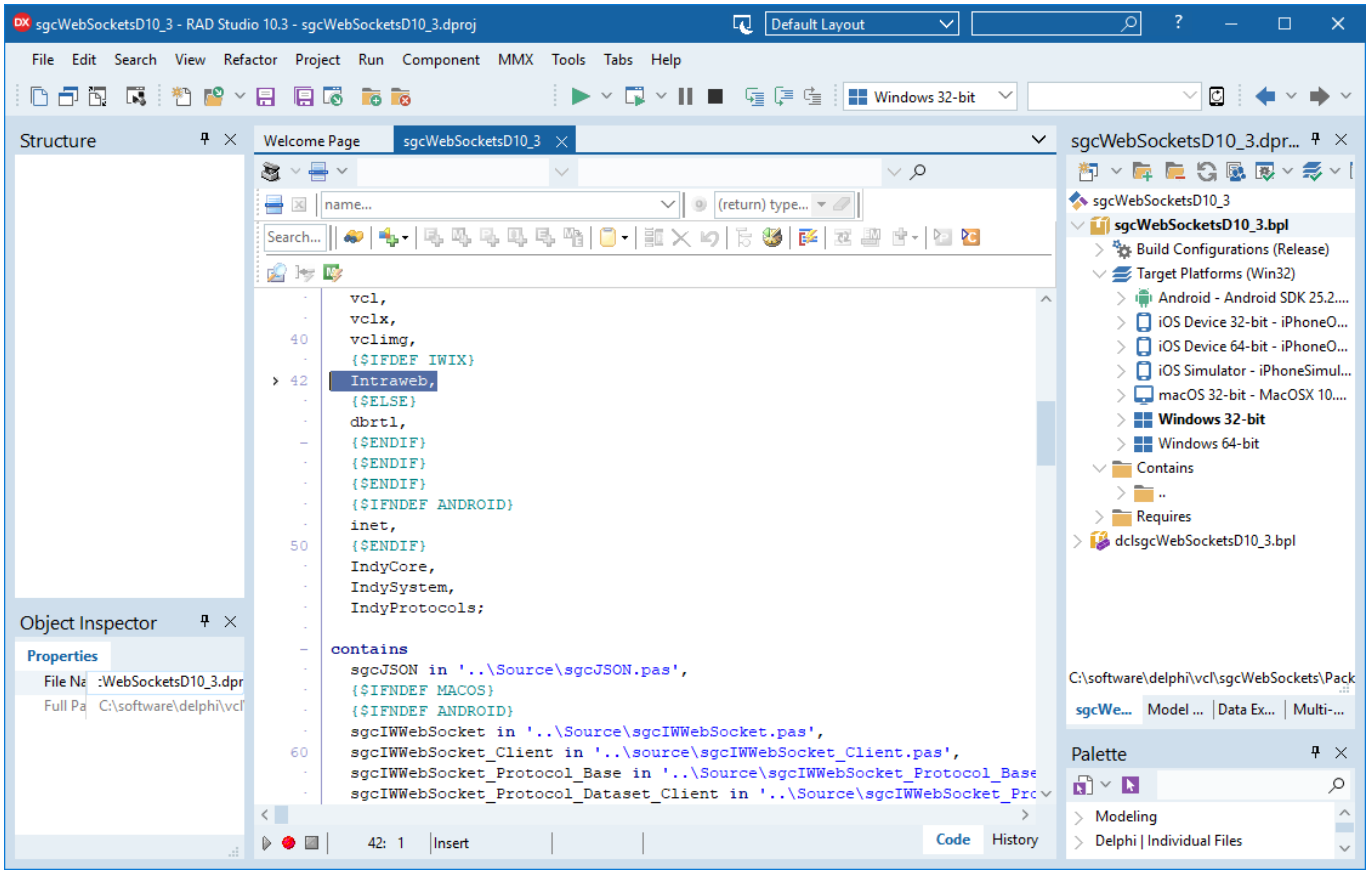
Sometimes you may get some errors installing components.

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...).

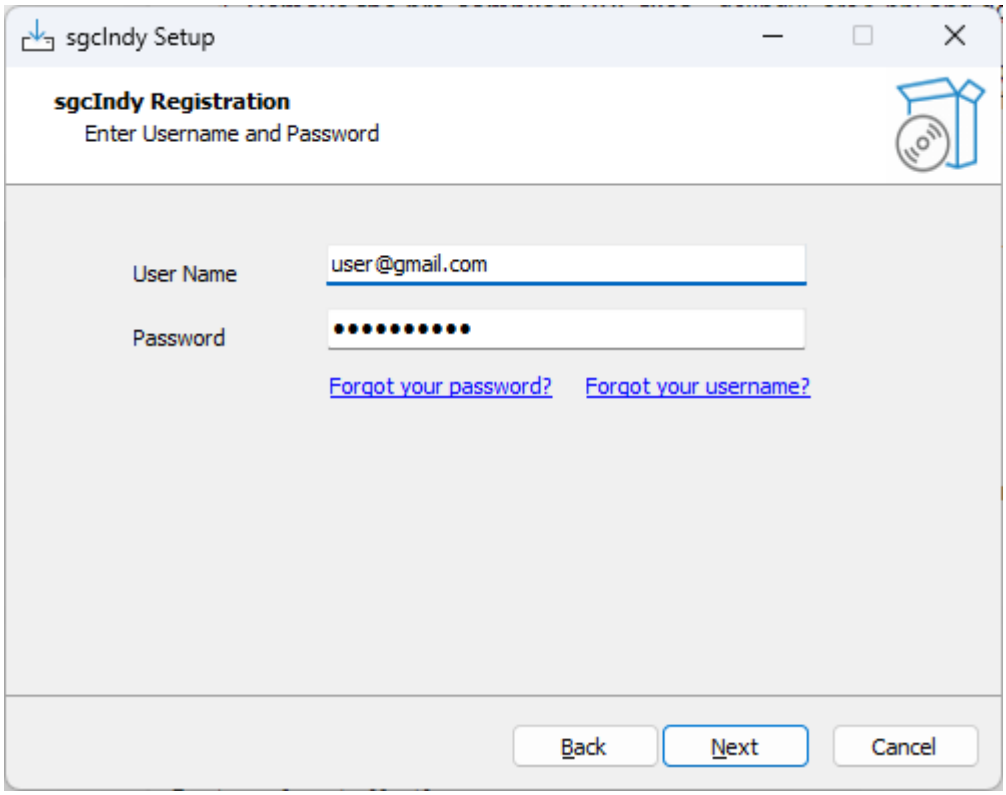
Install sgcIndy Package

Setup Installation

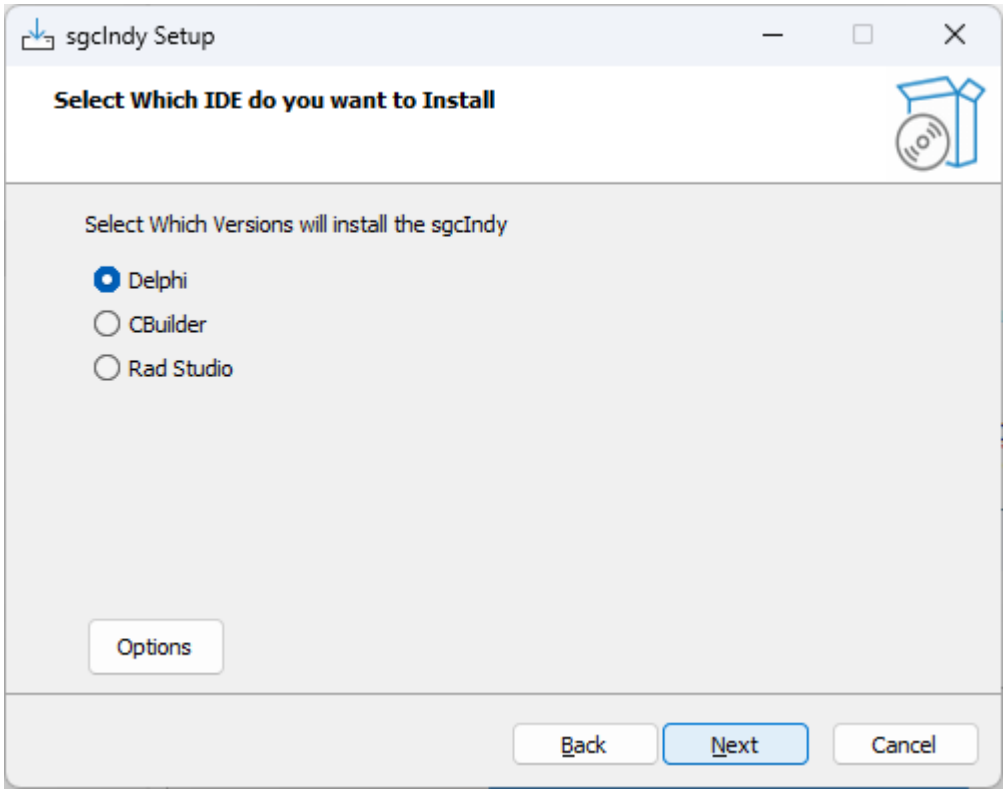
*Requires Windows Vista as minimum (Windows 2000, XP and Server 2003 are not supported).

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.

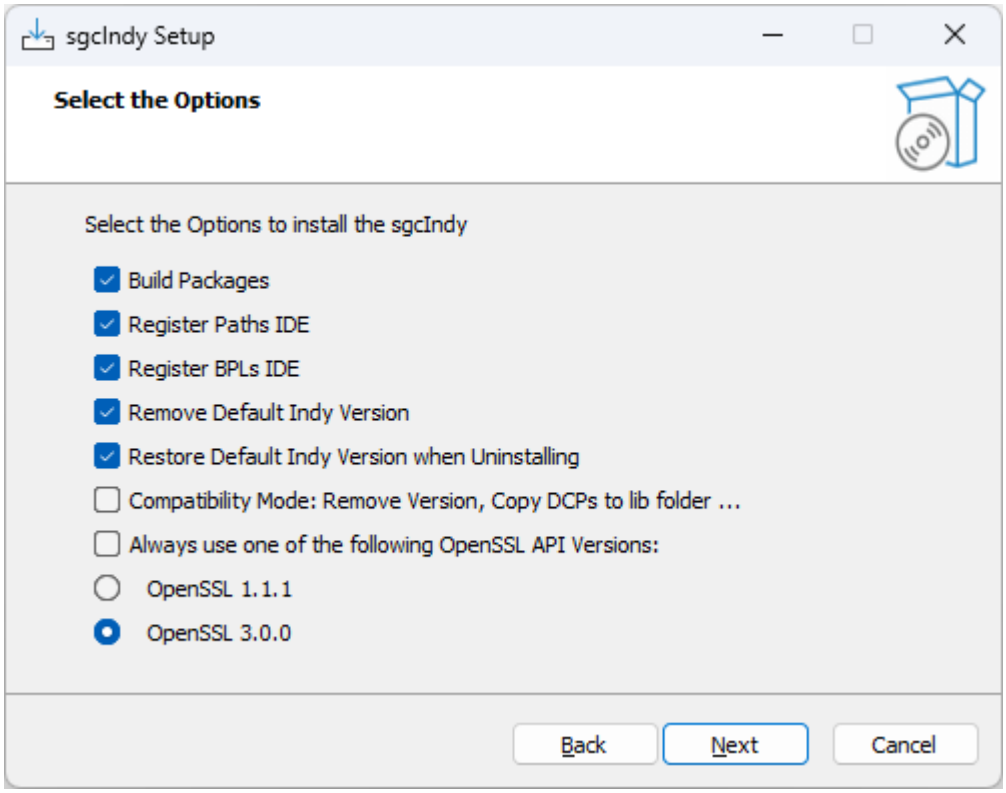
- Execute the Installer.
- First you must set your username/password of your private eSeGeCe account. This only must be entered one time, the next time you use the setup, the installer will read the latest value.



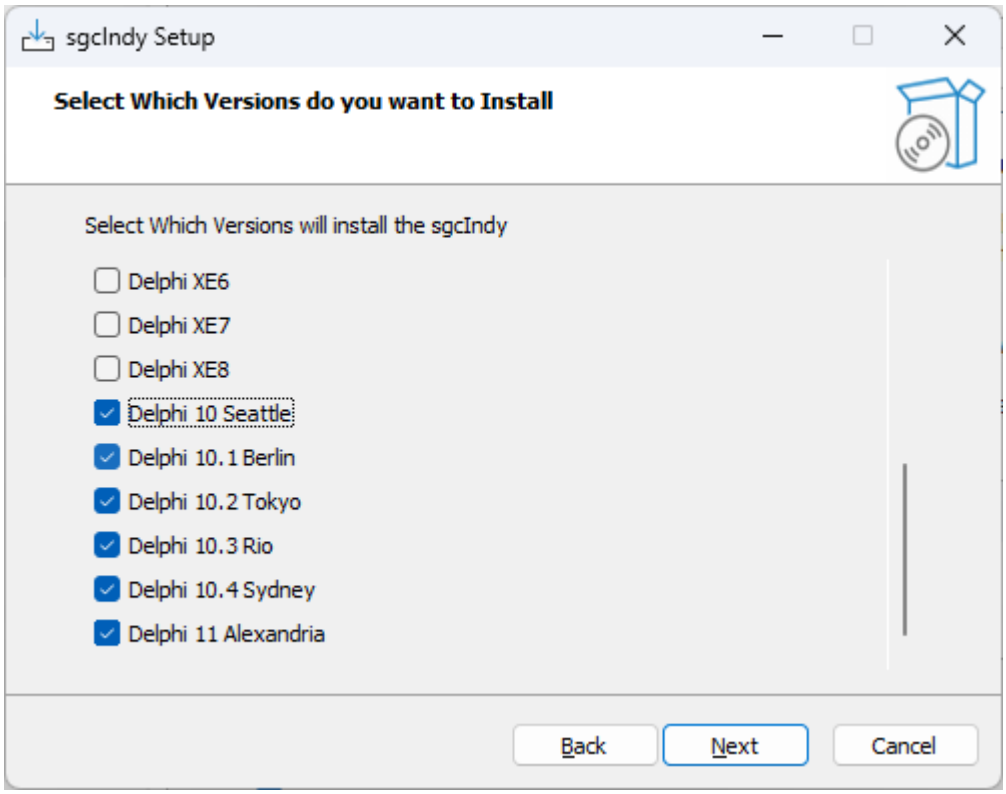
- If the user has logged in successfully, select whether you want to install in Delphi, CBuilder or Rad Studio IDE.



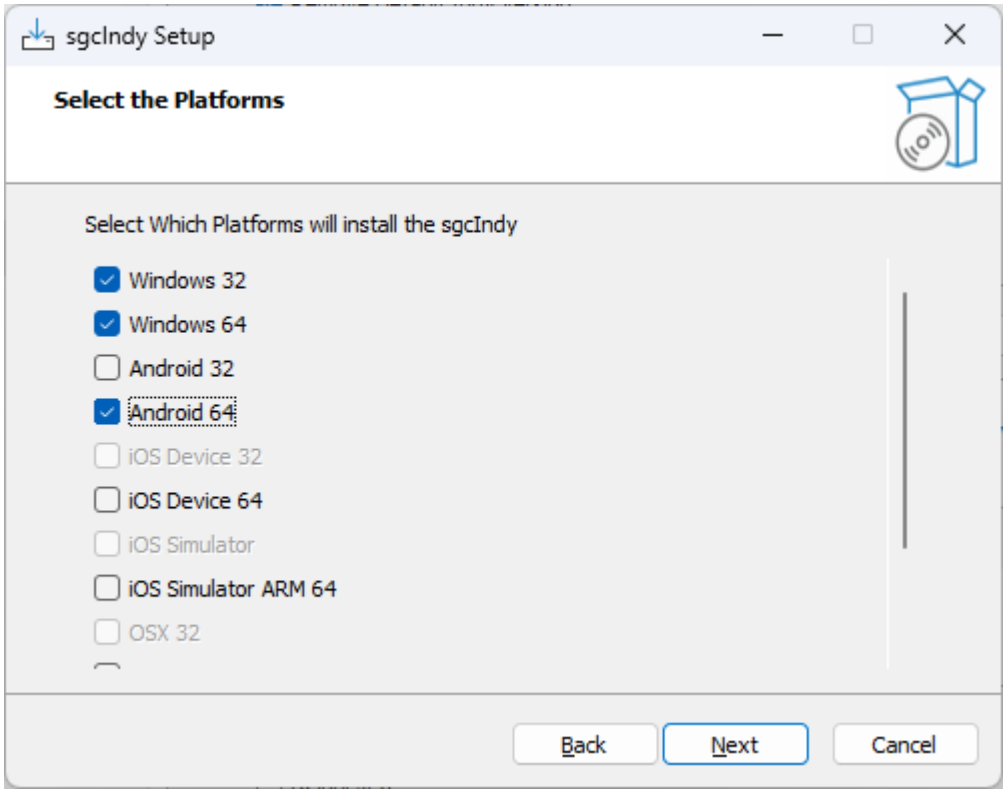
- There are some options that can be customized every time you use the installer, press the button **Options** to access these properties.
  - **Build Packages:** if selected, the installer will try to build the packages.
  - **Register Paths IDE:** if selected, the installer will register the required 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



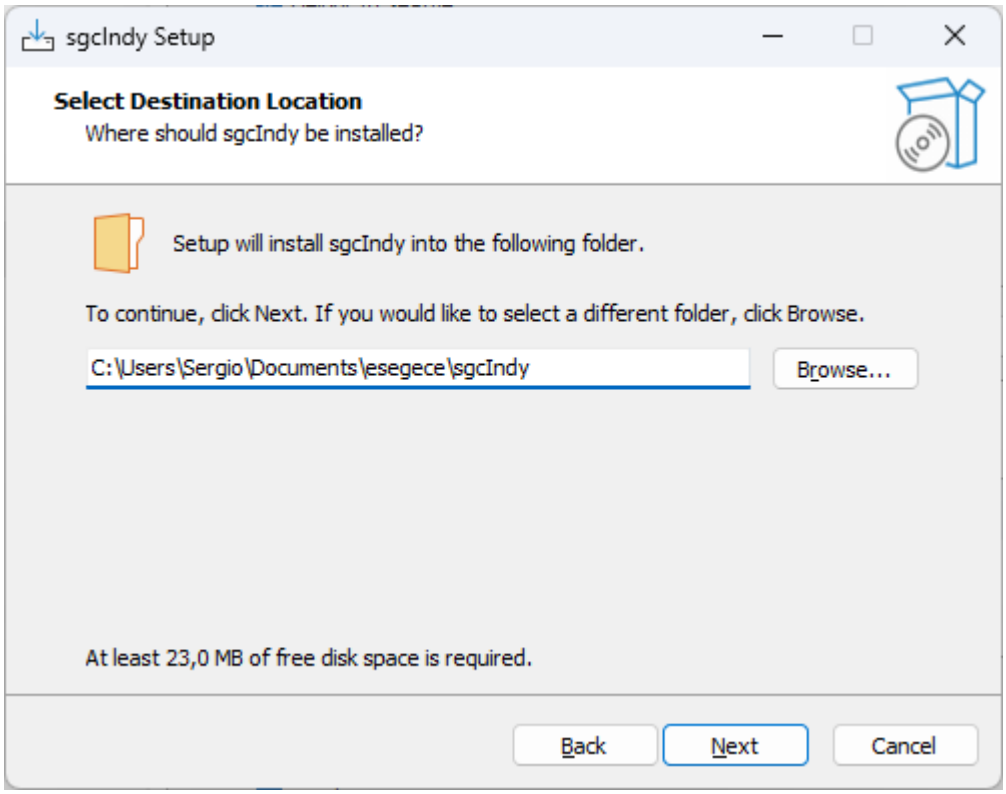
- Now you can select which IDE Versions you want to install. Only those IDE versions that the installer detect as installed, will be available.



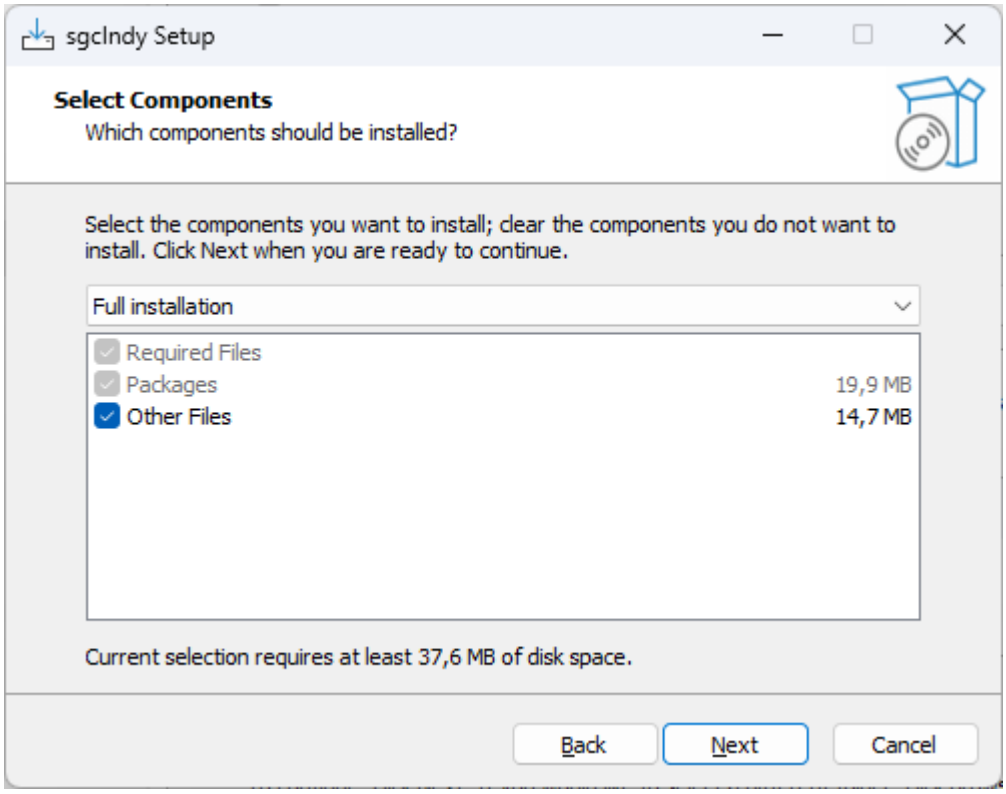
- Next step is select the Platforms.



- 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

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:

```
procedure OnMessage(Connection: TsgcWSConnection; const Text: string);
begin
  ShowMessage('Message Received From Client: ' + Text);
end;
```

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:

```
procedure OnCommandGet(AContext: TIdContext; ARequestInfo: TIdHTTPRequestInfo;
  AResponseInfo: TIdHTTPResponseInfo);
begin
  if ARequestInfo.Document = '/' then
  begin
    AResponseInfo.ContentText := '<html><head><title>Test Page</title></head><body></body></html>';
    AResponseInfo.ContentType := 'text/html';
    AResponseInfo.ResponseNo := 200;
  end;
end;
```

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 := t443;
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:

```
var
  oHTTP1: TsgcHTTP1Client;
begin
  oHTTP1 := TsgcHTTP1Client.Create(nil);
  Try
    ShowMessage(oHTTP1.Get('https://127.0.0.1'));
  Finally
    oHTTP1.Free;
  End;
end;
```

HTTP/2 Client

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

```
var
  oHTTP2: TsgcHTTP2Client;
begin
  oHTTP2 := TsgcHTTP2Client.Create(nil);
  Try
    ShowMessage(oHTTP2.Get('https://127.0.0.1'));
  Finally
    oHTTP2.Free;
  End;
end;
```

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.

```
procedure OnClientMessage(Connection: TsgcWSConnection; Text: String);
begin
```

```
  TThread.Queue(nil,
    procedure
        begin
          memo1.lines.add(Text);
        end);
end;
```

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

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.

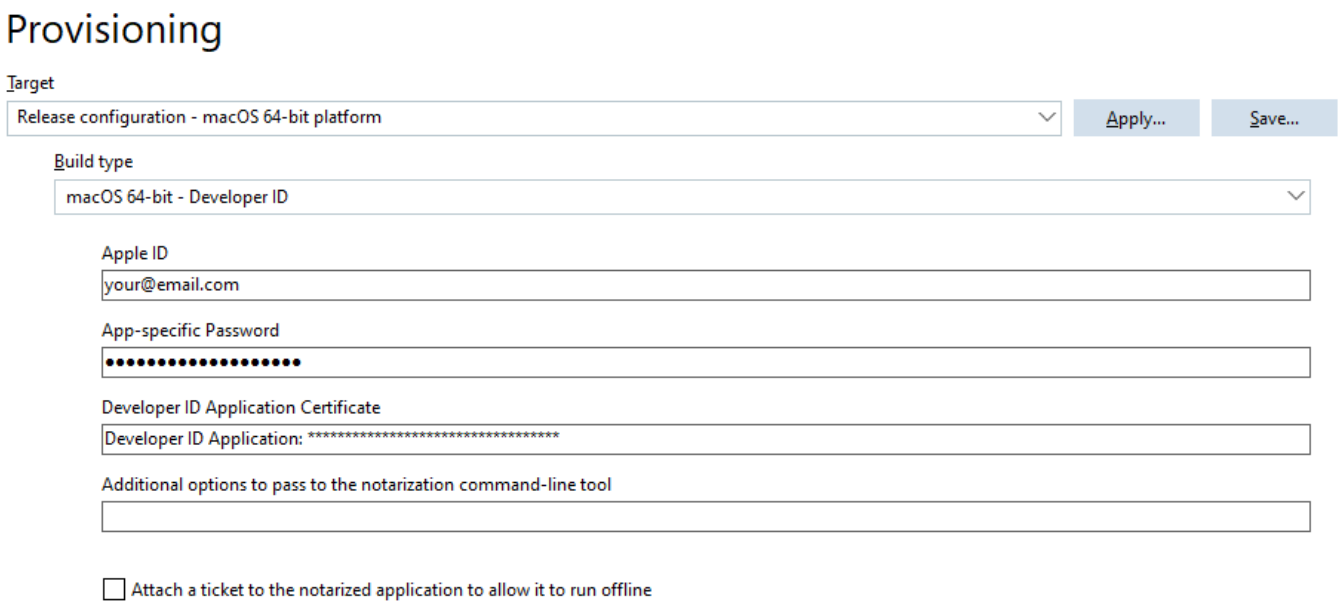
http://docwiki.embarcadero.com/RADStudio/Rio/en/MacOS_Notarization

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

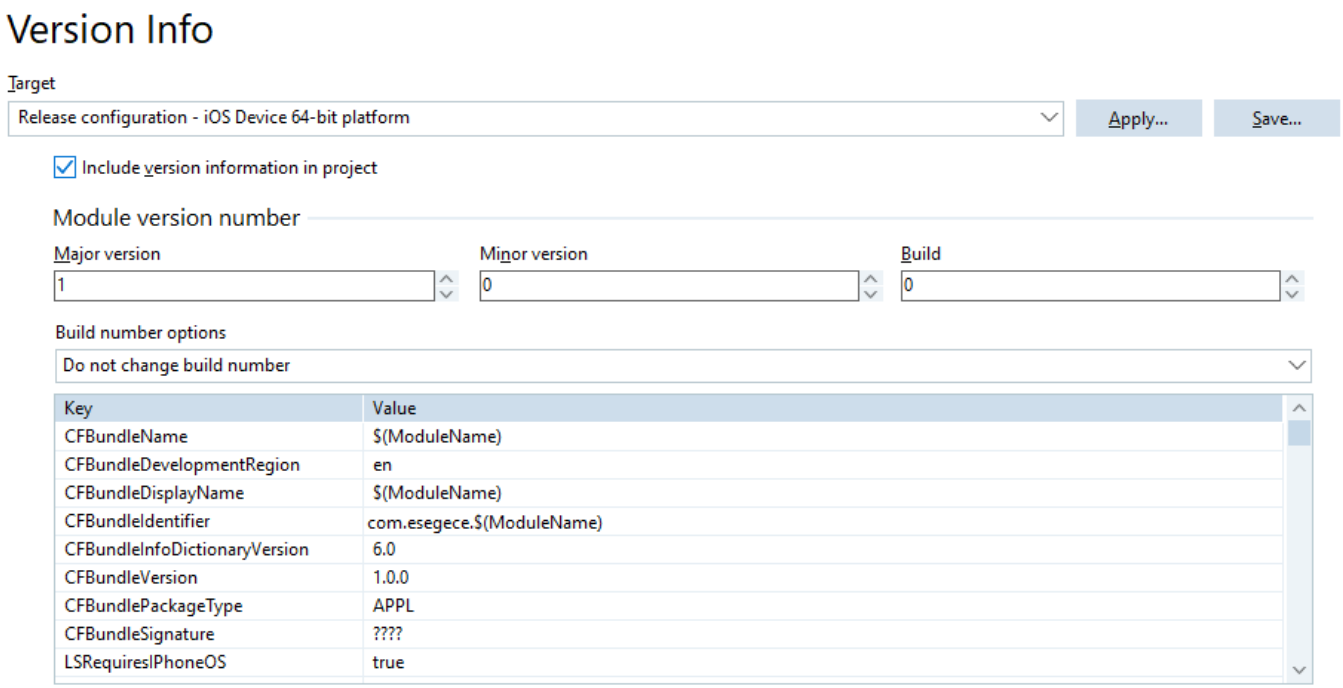
Obtain an iOS Development Certificate

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

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

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 = osIpDefaultFolder.
    - Server.SSLOptions.OpenSSL_Options.LibPath = osIpDefaultFolder.

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.

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(hIdSSL: TIdLibHandle; hIdCrypto: TIdLibHandle; const FailedLoadList: TStringL
```

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

```
TIdUnLoadSSLFuncsCallback = procedure();
```

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.

```
var
  EVP_PKEY_CTX_set_rsa_padding : function(ctx: PEVP_PKEY_CTX; pad: Integer): Integer; cdecl;

procedure DoOpenSSLLoadFuncsCallback(hIdSSL: TIdLibHandle; hIdCrypto:
    TIdLibHandle; const FailedLoadList: TStringList);
begin
  @EVP_PKEY_CTX_set_rsa_padding := LoadLibFunction(hIdCrypto, 'EVP_PKEY_CTX_set_rsa_padding');
end;

procedure DoOpenSSLUnLoadFuncsCallback;
begin
  @EVP_PKEY_CTX_set_rsa_padding := nil;
end;

IdOpenSSLSetLoadFuncsCallback(DoOpenSSLLoadFuncsCallback);
IdOpenSSLSetUnLoadFuncsCallback(DoOpenSSLUnLoadFuncsCallback);
```

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**.

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:

```
procedure WSServerConnect(Connection: TsgcWSConnection);
begin
  if Connection.URL = '/?id=1' then
    HandleThisParameter;
end;
```

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**.

```
procedure OnServiceStart(Sender: TService; var Started: Boolean);
begin
  FServer := TsgcWebSocketHTTPServer.Create(nil);
  FServer.Port := 80;
  FServer.Active := True;
end;
procedure OnServiceExecute(Sender: TService);
begin
  while not Terminated do
  begin
    ServiceThread.ProcessRequests(True);
  end;
end;
procedure OnServiceStop(Sender: TService; var Stopped: Boolean);
begin
  if Assigned(FServer) then
    FServer.Free;
end;
```

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.

```
//Client Example Code

unit sgcWebSocket_Protocol_Example_Client;

interface

{$I sgcVer.inc}
{$IFDEF SGC_PROTOCOLS}

uses
  sgcWebSocket_Protocol_Base_Client, Classes, sgcWebSocket_Classes;

type
  TsgcWSProtocol_Example_Client = class(TsgcWSProtocol_Client_Base)
  { from TsgcWSComponent }
  protected
    procedure DoEventConnect(aConnection: TsgcWSConnection); override;
    procedure DoEventMessage(aConnection: TsgcWSConnection; const Text: string);
        override;
    procedure DoEventDisconnect(aConnection: TsgcWSConnection; Code: Integer);
        override;
  { from TsgcWSComponent }
  public
    constructor Create(aOwner: TComponent); override;
  end;
{$ENDIF}

implementation

{$IFDEF SGC_PROTOCOLS}

constructor TsgcWSProtocol_Example_Client.Create(aOwner: TComponent);
begin
  inherited;
  // ... here add your protocol name
  FProtocol := 'MyProtocol';
end;

procedure TsgcWSProtocol_Example_Client.DoEventConnect(aConnection:
    TsgcWSConnection);
begin
  inherited;
  // ... add your own code when client connects to server
end;

procedure TsgcWSProtocol_Example_Client.DoEventDisconnect(aConnection:
    TsgcWSConnection; Code: Integer);
begin
  // ... add your own code when client disconnects from server
  inherited;
end;

procedure TsgcWSProtocol_Example_Client.DoEventMessage(aConnection:
    TsgcWSConnection; const Text: string);
begin
  // ... process messages received from server
  // ... you can send a message to server using WriteData('your message') method
end;
{$ENDIF}
```

```
end.

// Server Example Code

unit sgcWebSocket_Protocol_Example_Server;

interface

{$I sgcVer.inc}
{$IFDEF SGC_PROTOCOLS}

uses
  sgcWebSocket_Protocol_Base_Server, Classes, sgcWebSocket_Classes;

type
  TsgcWSProtocol_Example_Server = class(TsgcWSProtocol_Server_Base)
  { from TsgcWSComponent }
  protected
    procedure DoEventConnect(aConnection: TsgcWSConnection); override;
    procedure DoEventMessage(aConnection: TsgcWSConnection; const Text: string);
        override;
    procedure DoEventDisconnect(aConnection: TsgcWSConnection; Code: Integer);
        override;
  { from TsgcWSComponent }
  public
    constructor Create(aOwner: TComponent); override;
  end;
{$ENDIF}

implementation

{$IFDEF SGC_PROTOCOLS}
constructor TsgcWSProtocol_Example_Server.Create(aOwner: TComponent);
begin
  inherited;
  // ... here add your protocol name
  FProtocol := 'MyProtocol';
end;

procedure TsgcWSProtocol_Example_Server.DoEventConnect(aConnection:
    TsgcWSConnection);
begin
  inherited;
  // ... add your own code when a client connects to server
end;

procedure TsgcWSProtocol_Example_Server.DoEventDisconnect(aConnection:
    TsgcWSConnection; Code: Integer);
begin
  // ... add your own code when a client disconnects from server
  inherited;
end;

procedure TsgcWSProtocol_Example_Server.DoEventMessage(aConnection:
    TsgcWSConnection; const Text: string);
begin
  inherited;
  // ... process messages received from clients
  // ... you can answer to client using WriteData(aConnection.Guid, 'your message') method
  // ... you can send a message to all clients using BroadCast('your message') method
end;
{$ENDIF}

end.

//Implementation
// Once your custom subprotocol is implemented, then you only need to assign to your Client or Server webso

procedure InitalizeClient;
var
  oClient: TsgcWebSocketClient;
  oProtocol: TsgcWSProtocol_Example_Client;
begin
  oClient := TsgcWebSocketClient.Create(nil);
  oProtocol := TsgcWSProtocol_Example_Client.Create(nil);
  oProtocol.Client := oClient;
end;


procedure InitalizeServer;
var
  oServer: TsgcWebSocketServer;
  oProtocol: TsgcWSProtocol_Example_Server;
begin
  oClient := TsgcWebSocketServer.Create(nil);
  oProtocol := TsgcWSProtocol_Example_Server.Create(nil);
```

```
  oProtocol.Server := oServer;
end;
```

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:

```
procedure WSServerAuthentication(Connection: TsgcWSConnection; aUser, aPassword: string; var Authenticated: Boole
begin
  if (aUser = 'John') and (aPassword = '1234') then
```

```
    Authenticated := True;
end;
```

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

```
procedure OnServerSSLGetHandler(Sender: TObject; aType: TwsSSLHandler; var aSSLHandler:
TIdServerIOHandlerSSLBase);
begin
  aSSLHandler := TCustomSSLHandler.Create(nil);
  ...
end;
```

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.

```
procedure OnSSLAfterCreateHandler(Sender: TObject; aType: TwsSSLHandler; aSSLHandler:
TIdServerIOHandlerSSLBase);
begin
  TIdServerIOHandlerSSLOpenSSL(aSSLHandler).SSLOptions.Method := sslvTLSv1_2;
end;
```

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.

```
procedure OnBeforeHeartBeat(Sender: TObject; const Connection: TsgcWSConnection; var Handled: Boolean);
begin
  Connection.WriteData('ping');
  Handled := True;
end;
```

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;
```

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:

```
procedure WSServerCommandGet(AContext: TIdContext; ARequestInfo: TIdHTTPRequestInfo;
  AResponseInfo: TIdHTTPResponseInfo);
begin
  if ARequestInfo.Document = '/myfile.js' then
  begin
    AResponseInfo.ContentText := '<script type="text/javascript">alert("Hello!");</script>';
    AResponseInfo.ContentType := 'text/javascript';
    AResponseInfo.ResponseNo := 200;
  end
end;
```

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:

```
procedure OnServerConnect(Connection: TsgcWSConnection);
begin
  if desktop then
    TsgcWSConnectionServer(Connection).DoSubscribe('desktop');
end;
```

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.

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:

```
With WSServer.Bindings.Add do
begin
  Port := 80;
  IP := '127.0.0.1';
end;
With WSServer.Bindings.Add do
begin
  Port := 80;
  IP := '80.254.21.11';
end;
```

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.

```
oServer := TsgcWebSocketHTTPServer.Create(nil);
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.

```
procedure OnHTTPUploadBeforeSaveFileEvent(Sender: TObject; var aFileName: string; var aFilePath: string);
begin
  if aFileName = "test.jpg" then
    aFileName := "custom_test.jpg";
end;
```

OnHTTPUploadAfterSaveFile

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

```
procedure OnHTTPUploadBeforeSaveFileEvent(Sender: TObject; const aFileName: string; const aFilePath: string);
begin
  DoLog("File Received: " + aFileName);

end;
```

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...).

```
procedure OnHTTPUploadReadInputEvent(Sender: TObject; const aName: string; const aValue: string);
begin
  DoLog("Input value received: " + aName + ":" + aValue);

end;
```

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.

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.

Flash

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer

WebSockets are supported natively by a wide range of web browsers (please check http://caniuse.com/websockets), but there are some old versions that don't implement WebSockets (like Internet Explorer 6, 7, 8 or 9). You can enable **Flash Fallback** for all these browsers that don't implement WebSockets.

Almost all other or older browser support Flash installing Adobe Flash Player. To Support Flash connection, you need to **open port 843** on your server because Flash uses this port for security reasons to check for cross-domain-access. If port 843 is not reachable, waits 3 seconds and tries to connect to Server default port.

Flash is only applied if the Browser doesn't support WebSockets natively. So, if you enable Flash Fallback on the server side, and Web Browser supports WebSockets natively, it will still use WebSockets as transport.

To enable Flash Fallback, you need to access to **FallBack / Flash** property on the server and **enable** it. There are 2 properties more:

1. Domain: if you need to restrict flash connections to a single/multiple domains (by default all domains are allowed). Example: This will allow access to domain swf.example.com

```
swf.example.com
```

2. Ports: if you need to restrict flash connections to a single/multiple ports (by default all ports are allowed). Example: This will allow access to ports 123, 456, 457, and 458

```
123,456-458
```

Flash connections only support Text messages, binary messages are not supported.

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:
TMyClass = class
private
  FRegistered: Boolean;
  FUser: String;
public
  property Registered: Boolean read FRegistered write FRegistered;
  property User: String read FUser write FUser;
end;

// Then, when a new client connects, OnConnect Event, create a new TMyClass and Assign to Data:
procedure WSServerConnect(Connection: TsgcWSConnection);
begin
  Connection.Data := TMyClass.Create;
end;

// Every time a new message is received by the server, you can access your custom object
// using Connection.Data property.
procedure WSServerMessage(Connection: TsgcWSConnection; const Text: string);
begin
  if TMyClass(Connection.Data).Registered then
    DoSomeStuff();
end;

// When a connection is closed, you may free your object:
procedure TfrmServerChat.WSServerDisconnect(Connection: TsgcWSConnection; Code: Integer);
var
  oMyClass: TMyClass;
begin
  oMyClass := TMyClass(Connection.Data);
  if Assigned(oMyClass) then
  begin
    oMyClass.Free;
    Connection.Data := nil;
  end;
end;
```

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.

```
procedure OnConnect(Connection: TsgcWSConnection);
begin
  TsgcWebSocketServer1.Groups.Add('Room1', Connection);
end;
```

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;

procedure OnClientRemovedEvent(Sender: TObject; const aGroup: TsgcWSServerGroupItem;
  const aConnection: TsgcWSConnection);
begin
  aGroup.BroadCast('Client ' + aConnection.Guid + ' has disconnected');
end;
```

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;
```

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 should only be enabled if you want connections to always be processed in the same thread. When using IOCP, the requests are processed by a pool of threads, and every request (for the same connection) can be processed in different threads. If you want to handle every connection in the same thread set in WorkOpThreads the number of threads used to handle these requests. This impacts the performance of the server and it is only recommended to set a value greater than zero if you require this feature.

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.

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 2 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.

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;
```

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 only must be enabled if you want that connections are processed always in the same thread. When using EPOLL, the requests are processed by a pool of threads, and every request (for the same connection) can be processed in different threads. If you want to handle every connection in the same thread set in WorkOpThreads the number of threads used to handle these requests. This impacts in the performance of the server and it's only recommended to set a value greater of zero only if you require this feature.

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.

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 2 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.

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

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
procedure OnClientConnect(Connection: TsgcWSConnection);
var
  vProtocol: string;
begin
  vProtocol := TsgcWSConnectionClient(Connection).ALPNProtocol;
end;

// Server
procedure OnSSLALPNSelect(Sender: TObject; aProtocols: TStringList; var aProtocol: string);
var
  i: integer;
begin
  for i := 0 to aProtocols.count - 1 do
  begin
    if aProtocols[i] = 'h2' then
    begin
      aProtocol := 'h2';
      break;
    end;
  end;
end;
```

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:

```
procedure OnBeforeForwardHTTP(Connection: TsgcWSConnection; ARequestInfo: TIdHTTPRequestInfo;
  aForward: TsgcWSServerForwardHTTP);
begin
  if ARequestInfo.Document = '/internal' then
  begin
    aForward.Enabled := True;
    aForward.URL := 'http://localhost:8080';
  end;
end;
```

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

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.

```
procedure OnUnknownProtocol(Connection: TsgcWSConnection;  var Accept: Boolean);
begin
  Connection.Transport := trpTCP;
  Accept := True;
end;
```

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.

```
procedure OnWSClientConnect(Connection: TsgcWSConnection);
begin
  Connection.TCPEndOfFrameScanBuffer := eofScanAllBytes;
  Connection.AddTCPEndOfFrame(0);
  Connection.AddTCPEndOfFrame(10);
end;
```

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 := TsgcWebSocketClient.Create(nil);
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
oServer := TsgcWebSocketServer.Create(nil);
oServerBroker := TsgcWSPServer_Broker.Create(nil);
oServerBroker.Server := oServer;
oServerSGC := TsgcWSPServer_sgc.Create(nil);
oServerSGC.Broker := oServerBroker;
oServerFiles := TsgcWSPServer_files.create(nil);
oServerFiles.Broker := oServerBroker;
// ... client
oClient := TsgcWebSocketClient.Create(nil);
oClientBroker := TsgcWSPClient_Broker.Create(nil);
oClientBroker.Client := oClient;
oClientSGC := TsgcWSPClient_sgc.Create(nil);
oClientSGC.Broker := oClientBroker;
oClientFiles := TsgcWSPClient_files.create(nil);
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, 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' + #10 + 'data: Notification from server');

// If you need to specify an event name:
Connection.WriteData('event: notifications' + #10 + '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

```
procedure OnFragmented(Connection: TsgcWSConnection;  const Data: TMemoryStream; const OpCode: TOpcode; const Cor
begin
  ShowProgress(Data.Size);
  if not Continuation then
    SaveStream(Data);
end;
```

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 := TsgcWebSocketClient.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
...
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 := TsgcWebSocketClient.Create(nil);
...
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 := TsgcWebSocketClient.Create(nil);
...
if oClient.Connect(5000) then
  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) then
  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 := TsgcWebSocketClient.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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).

```
oClient := TsgcWebSocketClient.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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 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.

```
uses
```

```
sgcIdSSL, sgcSSL_SChannel_Indy, sgcSSL_SChannel;
```

```
var
  SSL: TsgcIdSSLIOHandlerSocketSChannel;
oClient := TsgcWebSocketClient.Create(nil);
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;
procedure OnSSLAfterCreateHandlerEvent(Sender: TObject; aType: TwsSSLHandler;
  aSSLHandler: TIdSSLIOHandlerSocketBase);
begin
  if aSSLHandler.ClassType = TsgcIdSSLIOHandlerSocketSChannel  then
    SSL := TsgcIdSSLIOHandlerSocketSChannel(aSSLHandler);
end;
procedure OnConnectEvent(Connection: TsgcWSConnection);
var
  oInfo: TsgcSChannelConnectionInfo;
begin
  if Assigned(SSL) then
  begin
    oInfo := SSL.GetInfo;
    if (oInfo.Protocol != tls1_2) then
      raise Exception.Create('Client cannot connect using TLS 1.2');
  end;
end;
```

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.

```
oStream := TMemoryStream.Create(nil);
Try
  ...
  TsgcWebSocketClient1.WriteData(oStream);
Finally
  oStream.Free;
End;
```

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.

```
procedure OnMessage(Connection: TsgcWSConnection; const Text: string);
begin
  ShowMessage('Message Received from Server: ' + Text);
end;
```

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.

```
procedure OnBinary(Connection: TsgcWSConnection; const Data: TMemoryStream);
var
  oBitmap: TBitmap;
begin
  oBitmap := TBitmap.Create;
  Try
    oBitmap.LoadFromStream(Data);
    Image1.Picture.Assign(oBitmap);
    Log(
      '#image uncompressed size: ' + IntToStr(Data.Size) +
      '. Total received: ' + IntToStr(Connection.RecBytes));
  Finally
    FreeAndNil(oBitmap);
  End;
end;
```

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 := TsgcWebSocketClient.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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...

```
procedure OnError(Connection: TsgcWSConnection; const Error: string);
begin
  WriteLn('#error: ' + Error);
end;
```

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.

```
procedure OnException(Connection: TsgcWSConnection; E: Exception);
begin
  WriteLn('#exception: ' + E.Message);
end;
```

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**.

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"

```
procedure OnHandshake(Connection: TsgcWSConnection; var Headers: TStringList);
begin
  Headers.Add('Client: sgcWebSockets');
end;
```

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 := TsgcWebSocketClient.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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

```
procedure OnTCPConnectEvent(aConnection: TsgcWSConnection; var Accept: Boolean);
begin
  aConnection.Transport := trpTCP;
  Accept := True;
end;
```

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 := TsgcWebSocketServer.Create(nil);
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 := TsgcWebSocketServer.Create(nil);
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 := TsgcWebSocketServer.Create(nil);
With oServer.Bindings.Add do
begin
  IP := '127.0.0.1';
  Port := 5555;
end;
With oServer.Bindings.Add do
begin
  IP := '80.58.25.40';
  Port := 5555;
end;
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.

```
procedure OnStartup(Sender: TObject);
begin
  WriteLn('#server started');
end;
```

OnShutdown

This event is fired after the server has stopped and no more connections are accepted.

```
procedure OnShutdown(Sender: TObject);
begin
  WriteLn('#server stopped');
end;
```

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 := TsgcWebSocketServer.Create(nil);
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 := TsgcWebSocketServer.Create(nil);
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 := TsgcWebSocketServer.Create(nil);
With oServer.Bindings.Add do
begin
  IP := '127.0.0.1';
  Port := 80;
end;
With oServer.Bindings.Add do
begin
  IP := '127.0.0.1';
  Port := 443;
end;
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;
```

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 := TsgcWebSocketServer.Create(nil);
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 := TsgcWebSocketServer.Create(nil);
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.

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.

```
function OnSSLVerifyPeerEvent(Sender: TObject; Certificate:
    TIdX509; AOk: Boolean; ADepth, AError: Integer; var Accept: Boolean);
begin
  // ... validate the certificate
  if Certificate_OK then
    Accept := True
  else
    Accept := False;
end;
```

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 := TsgcWebSocketServer.Create(nil);
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.

```
procedure DoClientIPAddresses;
var
  i: Integer;
  oList: TList;
  oConnection: TsgcWSConnectionServer;
begin
  oList := TsgcWebSocketHTTPServer1.LockList;
  Try
    for i := 0 to oList.Count - 1 do
    begin
      oConnection := TsgcWSConnectionServer(TIdContext(oList[i]).Data);
      ShowMessage(oConnection.IP + ':' + IntToStr(oConnection.Port));
    end;
  Finally
    TsgcWebSocketHTTPServer1.UnLockList;
  End;
end;
```

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.

```
procedure OnAuthentication(Connection: TsgcWSConnection; aUser, aPassword: string;
  var Authenticated: Boolean);
begin
  if ((aUser = 'user') and (aPassword = 'secret')) then
    Authenticated := true
  else
    Authenticated := false;
end;
```

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.

```
procedure OnUnknownAuthentication(Connection: TsgcWSConnection; AuthType, AuthData: string;
  var aUser, aPassword: string; var Authenticated: Boolean);
begin
  if AuthType = 'Bearer' then
  begin
    if AuthData = 'jwt_token' then
      Authenticated := true
    else
      Authenticated := false;
  end
  else
    Authenticated := false;
end;
```

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.

```
procedure OnConnect(Connection: TsgcWSConnection);
begin
  Connection.WriteData('Hello From Server');
end;
```

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', TMemoryStream.Create);
```

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.

```
procedure OnConnect(Connection: TsgcWSConnection);
begin
  Connection.WriteData(TMemoryStream.Create);
end;
```

Send a message to ALL connected clients

Call the **Broadcast()** method to send a binary message to all connected clients.

```
TsgcWebSocketServer1.Broadcast(TMemoryStream.Create);
```

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.

```
procedure OnMessage(Connection: TsgcWSConnection; const Text: string);
begin
  ShowMessage('Message Received from Client: ' + Text);
end;
```

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.

```
procedure OnBinary(Connection: TsgcWSConnection; const Data: TMemoryStream);
var
  oBitmap: TBitmap;
begin
  oBitmap := TBitmap.Create;
  Try
    oBitmap.LoadFromStream(Data);
    Image1.Picture.Assign(oBitmap);
    Log(
      '#image uncompressed size: ' + IntToStr(Data.Size) +
      '. Total received: ' + IntToStr(Connection.RecBytes));
  Finally
    FreeAndNil(oBitmap);
  End;
end;
```

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**.

```
procedure OnServerHandshake(Connection: TsgcWSConnection; var Headers: TStringList);
begin
  ShowMessage(TsgcWSConnectionServer(Connection).HeadersRequest.Text);
end;
```

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 — 33 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.

```
procedure OnCommandGet(AContext: TIdContext; ARequestInfo: TIdHTTPRequestInfo;
  AResponseInfo: TIdHTTPResponseInfo);
begin
  if ARequestInfo.Document = '/' then
  begin
    AResponseInfo.ContentText := '<html><head><title>Test Page</title></head><body></body></html>';
    AResponseInfo.ContentType := 'text/html';
    AResponseInfo.ResponseNo := 200;
  end;
end;
```

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.

```
procedure OnBeforeCommand(const aConnection: TsgcWSConnection; ARequestInfo: TIdHTTPRequestInfo; AResponseInfo: T
begin
  if aRequestInfo.Document = '/public' then
    aOptions := [hcoAuthorizedBasic];
end;
```

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.

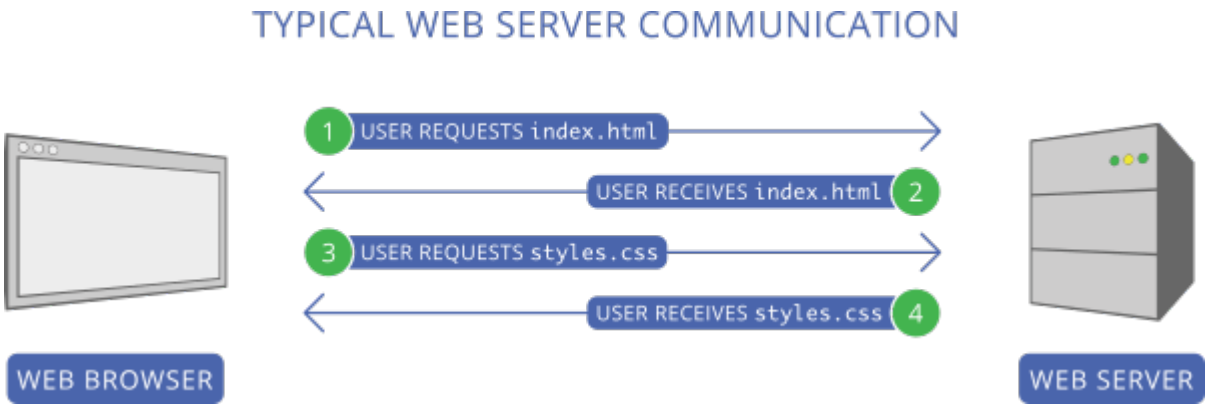
Set **HTTP2Options.Enabled = true** to allow the server to accept HTTP/2 protocol requests. The requests can be processed by the user exactly the same as with the HTTP/1.1 protocol, read more.

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.

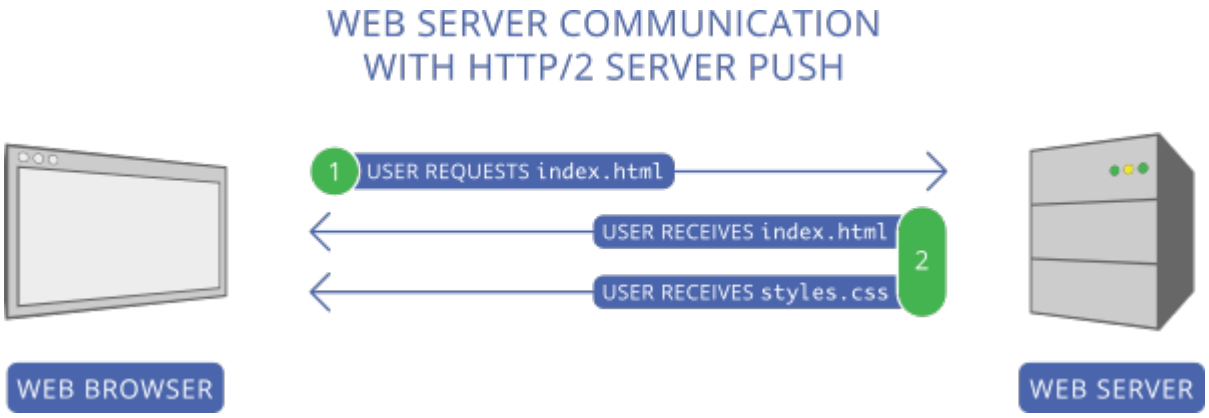
TsgcWebSocketHTTPServer | HTTP/2 Server Push

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

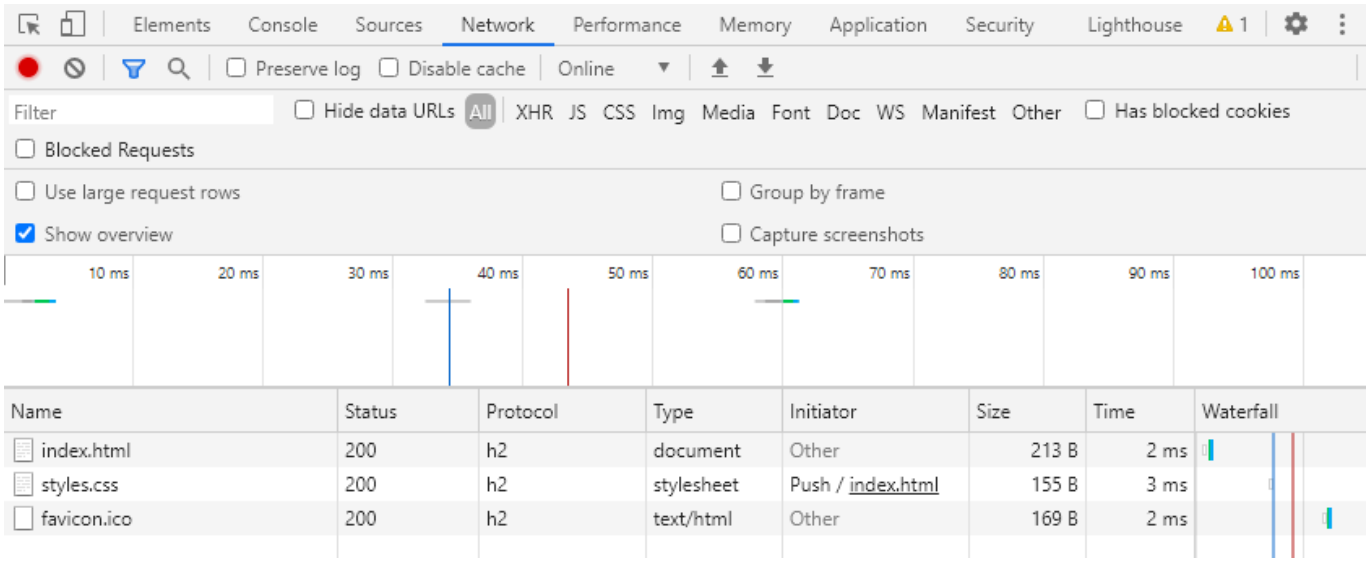
Following the screenshots above, you can configure your server so that every time there is a new request for the / index.html file, the server will send index.html and styles.css.

Use the method **PushPromiseAddPreLoadLinks** to associate each request with a push promise list.

```
server := TsgcWebSocketHTTPServer.Create(nil);
oLinks := TStringList.Create;
Try
  oLinks.Add('/styles.css');
  server.PushPromiseAddPreLoadLinks('/index.html', oLinks);
Finally
```

```
  oLinks.Free;
End;

      procedure OnCommandGet(AContext: TIdContext; ARequestInfo: TIdHTTPRequestInfo; AResponseInfo:
TIdHTTPResponseInfo);
begin
  if ARequestInfo.Document = '/index.html' then
  begin
    AResponseInfo.ContentText := '';
    AResponseInfo.ContentType := 'text/html';
    AResponseInfo.ResponseNo := 200;
  end
  else if ARequestInfo.Document = '/styles.css' then
  begin
    AResponseInfo.ContentText := '';
    AResponseInfo.ContentType := 'text/css';
    AResponseInfo.ResponseNo := 200;
  end;
end;
```

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.

```
procedure OnHTTP2BeforeAsyncRequest(Sender: TObject; Connection: TsgcWSConnection; const ARequestInfo: TIdHTTPRec
begin
  if ARequestInfo.Document = '/fast-request' then
    ASync := False;
end;
```

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

```
procedure OnCommandGet(AContext: TIdContext; ARequestInfo: TIdHTTPRequestInfo;
  AResponseInfo: TIdHTTPResponseInfo);
begin
  AResponseInfo.ContentStream := TStringStream.Create('');
  AResponseInfo.ContentType := 'text/html';
  AResponseInfo.ResponseNo := 404;
end;
```

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.

```
TsgcWebSocketHTTPServer1.SessionState := True;
TsgcWebSocketHTTPServer1.SessionTimeout := 600000;
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.

```
procedure OnCommandGet(AContext: TIdContext; ARequestInfo: TIdHTTPRequestInfo;
  AResponseInfo: TIdHTTPResponseInfo);
var
  vID: String;
  oSession: TIdHTTPSession;
begin
  if ARequestInfo.Document = '/' then
    AResponseInfo.ServeFile(AContext, 'yourpathhere\index.html')
  else
  begin
    // check if user is valid
    if not ((ARequestInfo.AuthUsername = 'user') and (ARequestInfo.AuthPassword = 'pass')) then
      AResponseInfo.AuthRealm := 'Authenticate'
    else
    begin
      // create a new session id with authentication data
      vID := ARequestInfo.AuthUsername + '_' + ARequestInfo.AuthPassword + '_' + ARequestInfo.RemoteIP;
```

```
      // search session
      oSession := TsgcWebSocketHTTPServer1.SessionList.GetSession(vID, ARequestInfo.RemoteIP);

      // create new session if not exists
      if not Assigned(oSession) then
        oSession := TsgcWebSocketHTTPServer1.SessionList.CreateSession(ARequestInfo.RemoteIP, vID);

      AResponseInfo.ContentText := '<html><head></head><body>Authenticated</body></html>';
      AResponseInfo.ResponseNo := 200;
    end;
  end;
end;
```

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

```
procedure OnServerCommandGet(AContext: TIdContext; ARequestInfo: TIdHTTPRequestInfo; AResponseInfo: TIdHTTPRespor
begin
  if ARequestInfo.Document = '/video.mp4' then
    IndyStreamFileVideo(Acontext, AResponseInfo, 'C:\videos\video.mp4')
  else
    AResponseInfo.ResponseNo := 404;
end;
```

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 := TsgcWebSocketHTTPServer.Create(nil);
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 := TsgcWebSocketHTTPServer.Create(nil);
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.

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

When set to True, the server calls `SetFileCompletionNotificationModes` on the request queue handle with the flags `FILE_SKIP_COMPLETION_PORT_ON_SUCCESS` and `FILE_SKIP_SET_EVENT_ON_HANDLE`. Overlapped I/O operations on the request queue that complete synchronously no longer post an extra completion packet to the I/O completion port.

Type: Boolean.
Default: False (no flag is set, preserving current behavior).
What it improves: eliminates a kernel-to-user-mode hop on the hot request path when the call returns `NO_ERROR` synchronously - the worker dispatches the completion inline on the calling thread instead of waiting for an IOCP packet. This is the pattern used by Microsoft's reference HTTP Server High Performance sample. The flag is intended to be paired with `OperatingMode = ompHighPerf`.

The API is resolved at runtime via `GetProcAddress` from kernel32. On Windows XP and earlier (Delphi 7 runtime target) the entry point does not exist - the wrapper returns False and the server runs without the flag set.

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, HighPerf dispatch with a widened pre-posted receive window, and inline-completion dispatch enabled.

```
uses
  sgcWebSocket_Server_HTTPAPI,
  sgcWebSocket_HTTPAPI_Server;   // TsgcHTTPAPIOperatingMode
var
  oServer: TsgcWSServer_HTTPAPI;
begin
  oServer := TsgcWSServer_HTTPAPI.Create(nil);
  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;
  // dispatch inline on sync-success completions; skip the IOCP round-trip
  oServer.FineTune.SkipIOCPOnSuccess := True;
  oServer.Active := True;
end;
```

For a typical internal or low-traffic API, leave every `FineTune` property at its default:

```
oServer := TsgcWSServer_HTTPAPI.Create(nil);
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.

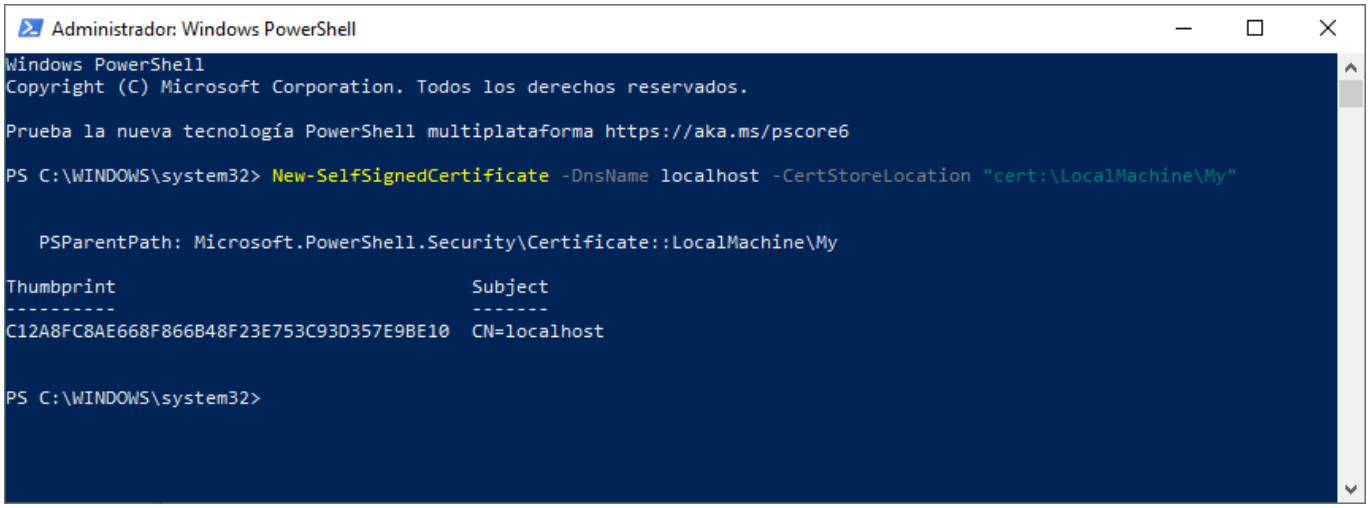
TsgcWebSocketServer_HTTPAPI | Self-Signed Certificates

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
```

```
procedure OnHTTPRequest(aConnection: TsgcWSConnection_HTTPAPI; const aRequestInfo: THttpServerRequest;
  var aResponseInfo: THttpServerResponse);
begin
  aResponseInfo.ResponseNo := 200;
  aResponseInfo.CustomHeaders := 'Access-Control-Allow-Origin: *' + #13#10 + 'Acces-Control-Allow-Methods: ' +
    'GET, POST, OPTIONS, PUT, PATCH, DELETE';
end;
```

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.

```
procedure OnHTTPRequest(aConnection: TsgcWSConnection_HTTPAPI;
        const aRequestInfo: THttpServerRequest;
        var aResponseInfo: THttpServerResponse);
begin
  if aRequestInfo.Method = 'GET' then
  begin
    if aRequestInfo.Document = '/test.html' then
        begin
          aResponseInfo.ResponseNo := 200;
          aResponseInfo.ContentText := 'OK';
          aResponseInfo.ContentType := 'text/html; charset=UTF-8';
        end
        else
          aResponseInfo.ResponseNo := 404;
  end
  else
    aResponseInfo.ResponseNo := 500;
end;
```

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.

```
procedure OnHTTPRequest(aConnection: TsgcWSConnection_HTTPAPI;
        const aRequestInfo: THttpServerRequest;
        var aResponseInfo: THttpServerResponse);
begin
  if aRequestInfo.Method = 'GET' then
  begin
    if aRequestInfo.Document = '/test.zip' then
        begin
          aResponseInfo.ResponseNo := 200;
          aResponseInfo.FileName := 'c:\download\test.zip';
          aResponseInfo.ContentType := 'application/zip';
        end
        else
          aResponseInfo.ResponseNo := 404;
  end
  else
    aResponseInfo.ResponseNo := 500;
end;
```

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.

```
procedure OnHTTPRequest(aConnection: TsgcWSConnection_HTTPAPI;
const aRequestInfo: THttpServerRequest; var aResponseInfo: THttpServerResponse);
var
  oStream: TFileStream;
  oRanges: TIdEntityRanges;
begin
  oStream := TFileStream.Create('test.pdf', fmOpenRead);
  oRanges := TIdEntityRanges.Create(nil);
  Try
    oRanges.Text := aRequestInfo.Range;
    aResponseInfo.ContentType := 'application/pdf';
    if oRanges.Count > 0 then
    begin
      aResponseInfo.ResponseNo := 206;
      aResponseInfo.AcceptRanges := 'bytes';
      aResponseInfo.ContentRangeStart := oRanges[0].StartPos;
      aResponseInfo.ContentRangeEnd := oRanges[0].EndPos;
      aResponseInfo.ContentRangeInstanceLength := oStream.Size;
      aResponseInfo.ContentStream := TIdHTTPRangeStream.Create(oStream,
        aResponseInfo.ContentRangeStart, aResponseInfo.ContentRangeEnd);
    end
    else
    begin
      aResponseInfo.ResponseNo := 200;
      aResponseInfo.ContentStream := oStream;
    end;
  Finally
    oRanges.Free;
  End;
end;
```

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 := TsgcWebSocketClient_WinHTTP.Create(nil);
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 — 16 properties available on the component.
- Methods — 8 public methods.
- Events — 9 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 fifteen 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, 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.

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).

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.

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.
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
procedure TForm1.ServerAuthentication(Connection: TsgcWSConnection;
  aUser, aPassword: String; var Authenticated: Boolean);
begin
  Authenticated := (aUser = 'admin') and (aPassword = 'secret');
  if not Authenticated then
    sgcWebSocketFirewall1.RegisterFailedAttempt(Connection.IP);
end;

// Optional: Handle firewall events for logging
procedure TForm1.FirewallViolation(Sender: TObject; const aIP: string;
  const aViolationType: TsgcFirewallViolationType; const aDetails: string);
begin
  Log('Firewall violation from ' + aIP + ': ' + aDetails);
end;
```

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 — 15 properties available on the component.
- Methods — 20 public methods.
- Events — 4 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.

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:

```
procedure TForm1.ServerAuthentication(Connection: TsgcWSConnection;
  aUser, aPassword: String; var Authenticated: Boolean);
begin
  Authenticated := CheckCredentials(aUser, aPassword);
  if not Authenticated then
    sgcWebSocketFirewall1.RegisterFailedAttempt(Connection.IP);
end;
```

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') then
  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:

```
procedure TForm1.FirewallViolation(Sender: TObject; const aIP: string;
  const aViolationType: TsgcFirewallViolationType; const aDetails: string);
begin
  case aViolationType of
    fvSQLInjection:
      LogWarning('SQL injection attempt from ' + aIP + ': ' + aDetails);
    fvXSS:
      LogWarning('XSS attempt from ' + aIP + ': ' + aDetails);
  end;
end;
```

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 may appear to come from the proxy IP. Configure your proxy to pass the real client IP (e.g., X-Forwarded-For header) and ensure the server resolves it correctly.

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;

with sgcWSRateLimiter1.PerEndpoint.Rules.Add as TsgcRateLimitRuleItem do
begin
  Name := 'expensive-report';
  Pattern := '*/api/v1/expensive-report*';
  Strategy := rlsSliding;
  MaxRequests := 10;
  WindowSec := 60;
end;

with sgcWSRateLimiter1.PerEndpoint.Rules.Add as TsgcRateLimitRuleItem do
begin
  Name := 'status';
  Pattern := '*/api/v1/status*';
  Strategy := rlsTokenBucket;
  MaxRequests := 1000;
  WindowSec := 60;
end;
```

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;

with sgcWSRateLimiter1.Quotas.Quotas.Add as TsgcRateLimitQuotaItem do
begin
  Name := 'free-tier-monthly';
  Scope := qscPerKey;
  Period := qpMonth;
  Limit := 10000;
end;
```

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
procedure TForm1.StatsTimer(Sender: TObject);
begin
  LabelRequests.Caption := Format('Requests: %d', [sgcWSRateLimiter1.Stats.TotalRequests]);
  LabelThrottled.Caption := Format('Throttled: %d', [sgcWSRateLimiter1.Stats.TotalThrottled]);
  LabelActiveKeys.Caption := Format('Active keys: %d', [sgcWSRateLimiter1.Stats.ActiveKeys]);
  LabelUptime.Caption := Format('Uptime: %d s', [sgcWSRateLimiter1.Stats.Uptime]);
end;
```

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;
with sgcWSRateLimiter1.PerEndpoint.Rules.Add as TsgcRateLimitRuleItem do
begin
  Name := 'expensive-report';
  Pattern := '*/api/v1/expensive-report*';
  Strategy := rlsSliding;
  MaxRequests := 10;
  WindowSec := 60;
end;

// 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;
with sgcWSRateLimiter1.Quotas.Quotas.Add as TsgcRateLimitQuotaItem do
begin
  Name := 'free-tier-monthly';
  Scope := qscPerKey;
  Period := qpMonth;
  Limit := 10000;
end;

// 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
procedure TForm1.RateLimiterThrottled(Sender: TObject;
  const aKey, aReason: string; var Allow: Boolean);
begin
  Log('Throttled ' + aKey + ': ' + aReason);
end;

// Handle quota exhausted events
procedure TForm1.RateLimiterQuotaExceeded(Sender: TObject;
  const aKey, aQuotaName: string);
begin
  Log('Quota ' + aQuotaName + ' exceeded for ' + aKey);
end;
```

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.

csHalfOpen	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
with sgcWSCircuitBreaker1.PerEndpoint.Items.Add as TsgcCircuitBreakerEndpointItem do
begin
  Pattern := 'api.openai.com';
  Enabled := True;
  OverrideThresholds.Enabled := True;
  OverrideThresholds.FailureCount := 10;
  OverrideThresholds.FailureRatePercent := 70;
  OverrideRecovery.CooldownSec := 60;
end;
```

```
// Stripe - payment critical: open fast, recover slow
with sgcWSCircuitBreaker1.PerEndpoint.Items.Add as TsgcCircuitBreakerEndpointItem do
begin
  Pattern := 'api.stripe.com';
  Enabled := True;
  OverrideThresholds.Enabled := True;
  OverrideThresholds.FailureCount := 3;
  OverrideThresholds.FailureRatePercent := 30;
  OverrideRecovery.CooldownSec := 120;
end;
```

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
procedure TForm1.MetricsTimer(Sender: TObject);
begin
  LabelCalls.Caption := Format('Calls: %d (ok %d / fail %d / rejected %d)',
    [sgcWSCircuitBreaker1.Metrics.TotalCalls,
     sgcWSCircuitBreaker1.Metrics.TotalSuccesses,
     sgcWSCircuitBreaker1.Metrics.TotalFailures,
     sgcWSCircuitBreaker1.Metrics.TotalRejected]);
  LabelOpen.Caption := Format('Open breakers: %d',
    [sgcWSCircuitBreaker1.Metrics.CurrentOpenBreakers]);
  LabelLatency.Caption := Format('Avg latency: %.1f ms',
    [sgcWSCircuitBreaker1.Metrics.AverageLatencyMs]);
end;
```

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;
with sgcWSCircuitBreaker1.PerEndpoint.Items.Add as TsgcCircuitBreakerEndpointItem do
begin
  Pattern := 'api.stripe.com';
  OverrideThresholds.Enabled := True;
  OverrideThresholds.FailureCount := 3;
  OverrideRecovery.CooldownSec := 120;
end;

// 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
  Response := sgcAI_OpenAI1.Get('https://api.openai.com/v1/models');
except
  on E: Exception do
    if Pos('Circuit breaker open', E.Message) > 0 then
      Log('Circuit open for api.openai.com - using fallback')
    else
      Log('Call failed: ' + E.Message);
end;

// Handle state changes
procedure TForm1.CircuitStateChange(Sender: TObject; const aKey: string;
  aOldState, aNewState: TsgcCircuitState);
begin
  Log(Format('Circuit %s: %d -> %d', [aKey, Ord(aOldState), Ord(aNewState)]));
end;

// Handle fallback responses
procedure TForm1.CircuitFallback(Sender: TObject; const aKey: string;
  var aResponse: string);
begin
  Log('Fallback used for ' + aKey);
end;
```

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;

with sgcWSAPIKeyManager1.Scopes.Scopes.Add as TsgcAPIKeyScopeItem do
begin
  Name := 'read:orders';
  Description := 'List and read order history';
end;

with sgcWSAPIKeyManager1.Scopes.Scopes.Add as TsgcAPIKeyScopeItem do
begin
  Name := 'write:shipments';
  Description := 'Create and update shipment records';
end;
```

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
procedure TForm1.StatsTimer(Sender: TObject);
begin
  LabelActive.Caption := Format('Active: %d', [sgcWSAPIKeyManager1.Stats.ActiveKeys]);
  LabelRevoked.Caption := Format('Revoked: %d', [sgcWSAPIKeyManager1.Stats.RevokedKeys]);
  LabelValidations.Caption := Format('Validations: %d', [sgcWSAPIKeyManager1.Stats.TotalValidations]);
  LabelFailures.Caption := Format('Failures: %d', [sgcWSAPIKeyManager1.Stats.TotalValidationFailures]);
end;
```

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;
with sgcWSAPIKeyManager1.Scopes.Scopes.Add as TsgcAPIKeyScopeItem do
begin Name := 'read:orders';     Description := 'Read orders'; end;
with sgcWSAPIKeyManager1.Scopes.Scopes.Add as TsgcAPIKeyScopeItem do
begin Name := 'write:orders';    Description := 'Create orders'; end;

// 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)
var
  LKey, LNewKey: string;
begin
  LKey := sgcWSAPIKeyManager1.IssueKey('customer-123',
    TArray<string>.Create('read:orders', 'write:orders'), 30 * 86400);
  ShowMessage('Deliver to customer: ' + LKey);
end;

// Authorize every incoming connection in OnConnect
procedure TForm1.ServerConnect(Connection: TsgcWSConnection);
begin
  if not sgcWSAPIKeyManager1.IsRequestAuthorized(
    Connection.HeadersRequest.Text, Connection.URL, Connection.IP, 'read:orders') then
    Connection.Disconnect;
end;

// 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.

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 := TsgcIWWebSocketClient.Create(nil);
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.

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.

Transport: returns the transport type of connection:

- **trpRFC6455:** a normal WebSocket connection.
- **trpHixie76:** a WebSocket connection using draft WebSocket spec.
- **trpFlash:** a WebSocket connection using Flash as FallBack.

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 AppRTC: is a webrtc demo application developed by Google and Mozilla, it enables both browsers to “talk” to each other using the WebRTC API.

3. Protocol WebRTC: open source project aiming to enable the web with Real-Time Communication (RTC) capabilities.

4. Protocol Files: implemented using binary messages, provides support for sending files: packet size, authorization, QoS, message acknowledgement and more.

5. Protocol SGC: implemented using JSON-RPC 2.0 messages, provides the following patterns: RPC, PubSub, Transactional Messages, Messages Acknowledgment and more.

6. Protocol Dataset: inherits from Default Protocol, can send dataset changes (new record, save record or delete a record) from the server to clients.

7. 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...

8. Protocol WAMP 1.0: open WebSocket subprotocol that provides two asynchronous messaging patterns: RPC and PubSub.

9. Protocol WAMP 2.0: open WebSocket subprotocol that provides two asynchronous messaging patterns: RPC and PubSub.

10. 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.

10.1 STOMP for RabbitMQ: client for RabbitMQ Broker.

10.2 STOMP for ActiveMQ: client for ActiveMQ Broker.

11. 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.

12. 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.

13. 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 assign 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 := TsgcWebSocketClient.Create(nil);
oClient.Host := 'test.mosquitto.org';
oClient.Port := 8080;
oMQTT := TsgcWSPClient_MQTT.Create(nil);
oMQTT.Client := oClient;
oClient.Active := True;

procedure OnMQTTConnect(Connection: TsgcWSConnection; const Session: Boolean; const ReasonCode: Integer;
  const ReasonName: string; const ConnectProperties: TsgcWSMQTTCONNACKProperties);
begin
  oMQTT.Subscribe('topic1');
end;
```

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.

```
procedure OnMQTTBeforeConnect(Connection: TsgcWSConnection; var aCleanSession: Boolean;
  var aClientIdentifier: string);
begin
  aClientIdentifier := 'your client id';
end;
```

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 := TsgcWSPClient_MQTT.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
oClient.Host := 'test.mosquitto.org';
oClient.Port := 8080;
oMQTT := TsgcWSPClient_MQTT.Create(nil);
oMQTT.Client := oClient;
oClient.Active := True;
```

MOSQUITTO MQTT WebSockets TLS

```
oClient := TsgcWebSocketClient.Create(nil);
oClient.Host := 'test.mosquitto.org';
oClient.Port := 8081;
oClient.TLS := True;
oClient.TLSOptions.Version := tls1_2;
oMQTT := TsgcWSPClient_MQTT.Create(nil);
oMQTT.Client := oClient;
oClient.Active := True;
```

MOSQUITTO MQTT Plain TCP

```
oClient := TsgcWebSocketClient.Create(nil);
oClient.Host := 'test.mosquitto.org';
oClient.Port := 1883;
oClient.Specifications.RFC6455 := False;
oMQTT := TsgcWSPClient_MQTT.Create(nil);
oMQTT.Client := oClient;
oClient.Active := True;
```

MOSQUITTO MQTT Plain TCP TLS

```
oClient := TsgcWebSocketClient.Create(nil);
oClient.Host := 'test.mosquitto.org';
oClient.Port := 8083;
oClient.Specifications.RFC6455 := False;
oClient.TLS := True;
oClient.TLSOptions.Version := tls1_2;
oMQTT := TsgcWSPClient_MQTT.Create(nil);
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.

```
procedure TfrmWebSocketClient.MQTTMQTTBeforeConnect(Connection: TsgcWSConnection;
  var aCleanSession: Boolean; var aClientIdentifier: string);
begin
  aCleanSession := false;
  aClientIdentifier := 'previous client id';
end;

procedure OnMQTTConnect(Connection: TsgcWSConnection; const Session: Boolean;
  const ReasonCode: Integer; const ReasonName: string; const ConnectProperties: TsgcWSMQTTCONNACKProperties);
begin
  if Session then
    WriteLn('Session resumed')
  else
    WriteLn('New Session');
end;
```

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 := TsgcWebSocketClient.Create(nil);
oClient.Host := 'test.mosquitto.org';
oClient.Port := 8080;
oMQTT := TsgcWSPClient_MQTT.Create(nil);
oMQTT.Client := oClient;
oClient.Active := True;

procedure OnMQTTConnect(Connection: TsgcWSConnection; const Session: Boolean; const ReasonCode: Integer;
  const ReasonName: string; const ConnectProperties: TsgcWSMQTTCONNACKProperties);
begin
  oMQTT.Subscribe('topic1');
end;
```

Publish Message

Publish a message to all subscribers of "topic1"

```
oClient := TsgcWebSocketClient.Create(nil);
oClient.Host := 'test.mosquitto.org';
oClient.Port := 8080;
oMQTT := TsgcWSPClient_MQTT.Create(nil);
oMQTT.Client := oClient;
oClient.Active := True;

procedure OnMQTTConnect(Connection: TsgcWSConnection; const Session: Boolean; const ReasonCode: Integer;
  const ReasonName: string; const ConnectProperties: TsgcWSMQTTCONNACKProperties);
begin
  oMQTT.Publish('topic1', 'Hello Subscribers topic1');
end;
```

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 := TsgcWSMQTTSubscribe_Properties.Create;
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

```
procedure OnMQTTPublish(Connection: TsgcWSConnection; aTopic, aText: string;
  PublishProperties: TsgcWSMQTTPublishProperties);
begin
  WriteLn('Topic: ' + aTopic + '. Message: ' + aText);
end;
```

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)

```
procedure OnMQTTPublishEx(Connection: TsgcWSConnection; aTopic: string;
  aData: TsgcWSMQTTPublishData; PublishProperties: TsgcWSMQTTPublishProperties);
begin
  // read as string
  WriteLn('Topic: ' + aTopic + '. Message: ' + aData.Value);
  // read as bytes
  WriteLn('Bytes Length: ' + IntToStr(Length(aData.Bytes)));
  // read as stream
  WriteLn('Stream Size: ' + IntToStr(aData.Stream.Size));
end;
```

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') then
  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.

```
oAMQP := TsgcWSPClient_AMQP.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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');

procedure OnAMQPGetOk(Sender: TObject; const aChannel: string;
  const aGetOk: TsgcAMQPFramePayload_Method_BasicGetOk; const aContent: TsgcAMQPMessageContent)
begin
  DoLog('#AMQP_basic_GetOk: ' + aChannel + ' ' + IntToStr(aGetOk.MessageCount) + ' ' + aContent.Body.AsString
end;
```

Pushed By Server

```
oAMQP.Consume('channel_name', 'queue_name');

procedure OnAMQPGetOk(Sender: TObject; const aChannel, aConsumerTag: string)
begin
  DoLog('#AMQP_basic_GetOk: ' + aChannel + ' ' + IntToStr(aGetOk.MessageCount) + ' ' + aContent.Body.AsString
end;
```

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 := TsgcWSPClient_AMQP.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
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 := TsgcWSPClient_AMQP.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
oAMQP.Client := oClient;
oClient.Specifications.RFC6455 := false;
oClient.Host := 'www.esegece.com';
oClient.Port := 5672;
oClient.Active := True;

procedure OnAMQPAuthentication(Sender: TObject; aMechanisms: TsgcAMQPAuthentications; var Mechanism: TsgcAMQPAuth
  var User, Password: string);
begin
  User := 'user_value';
  Password := 'password_value';
end;
```

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');

procedure OnAMQPChannelOpen(Sender: TObject; const aChannel: string);
begin
  DoLog('#AMQP_channel_open: ' + aChannel);
end;
```

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') then
  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');

procedure OnAMQPExchangeDeclare(Sender: TObject; const aChannel, aExchange: string);
begin
  DoLog('#AMQP_exchange_declare: [' + aChannel + '] ' + aExchange);
end;
```

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') then
  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');

procedure OnAMQPQueueDeclare(Sender: TObject; const aChannel, aQueue: string;
  aMessageCount, aConsumerCount: Integer);
begin
  DoLog('#AMQP_queue_declare: [' + aChannel + '] ' + aQueue);
end;
```

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') then
  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.BindQueue('channel_name', 'queue_name', 'exchange_name', 'routing_key');

procedure OnAMQPQueueBind(Sender: TObject; const aChannel, aQueue, aExchange: string);
begin
  DoLog('#AMQP_queue_bind: [' + aChannel + '] ' + aQueue + ' -->-- ' + aExchange)
end;
```

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') then
  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!!!');

procedure OnAMQPBasicReturn(Sender: TObject; const aChannel: string;
  const aReturn: TsgcAMQPFramePayload_Method_BasicReturn;
  const aContent: TsgcAMQPMessageContent);
begin
  DoLog('#AMQP_basic_return: ' + aChannel + ' ' + IntToStr(aReturn.ReplyCode) + ' ' + aReturn.ReplyText + ' ' + a
end;
```

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');

procedure OnAMQPBasicDeliver(Sender: TObject;
  const aChannel: string;
  const aDeliver: TsgcAMQPFramePayload_Method_BasicDeliver;
  const aContent: TsgcAMQPMessageContent);
begin
  DoLog('#AMQP_basic_deliver: ' + aChannel + ' ' + aDeliver.ConsumerTag + ' ' +
    ' ' + aContent.Body.AsString);
end;
```

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');

procedure OnAMQPBasicCancelConsume(Sender: TObject; const aChannel: string; const aConsumerTag);
begin
  DoLog('#AMQP_basic_cancel_consume: ' + aChannel + ' ' + aConsumerTag);
end;
```

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');

procedure OnAMQPBasicGetOk(Sender: TObject; const aChannel: string;
  const aGetOk: TsgcAMQPFramePayload_Method_BasicGetOk;
  const aContent: TsgcAMQPMessageContent);
begin
  DoLog('#AMQP_basic_GetOk: ' + aChannel + ' ' + IntToStr(aGetOk.MessageCount) + ' ' + aContent.Body.AsString);
end;

procedure OnAMQPBasicGetEmpty(Sender: TObject; const aChannel: string);
begin
  DoLog('#AMQP_basic_GetEmpty: ' + aChannel);
end;
```

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);

procedure OnAMQPBasicQoS(Sender: TObject; const aChannel: string;
  const aQoS: TsgcAMQPFramePayload_Method_BasicQoS);
begin
  DoLog('#AMQP_basic_qos: ' + aChannel + ' ' + IntToStr(aQoS.PrefetchSize) + ' '
    + IntToStr(aQoS.PrefetchCount) + ' ' + BoolToStr(aQoS.Global));
end;
```

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');

procedure OnAMQPTransactionOk(Sender: TObject; const aChannel: string; aTransaction: TsgcAMQPTransaction);
begin
  case aTransaction of
    amqpTransactionSelect:
      DoLog('#AMQP_transaction_ok: [' + aChannel + '] select');
    amqpTransactionCommit:
      DoLog('#AMQP_transaction_ok: [' + aChannel + '] commit');
    amqpTransactionRollback:
      DoLog('#AMQP_transaction_ok: [' + aChannel + '] rollback');
  end;
end;
```

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 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 := TsgcWSPClient_AMQP1.Create(nil);
```

```
// Setting AMQP authentication options
oAMQP.AMQPOptions.Authentication.AuthType := amqp1authSASLPlain;
oAMQP.AMQPOptions.Authentication.Username := 'sgc';
oAMQP.AMQPOptions.Authentication.Password := 'sgc';
// Creating WebSocket client
oClient := TsgcWebSocketClient.Create(nil);
// Setting WebSocket specifications
oClient.Specifications.RFC6455 := False;
// Setting WebSocket client properties
oClient.Host := '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 **Create-Session** 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');
procedure AMQP1AMQPSessionOpen(Sender: TObject; const aSession: TsgcAMQP1Session; const aBegin: TsgcAMQP1Fram
begin
  ShowMessage('#session-open: ' + aSession.Id);
end;
```

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');
procedure procedure TfrmClientAMQP1.AMQP1AMQPLinkOpen(Sender: TObject; const aSession: TsgcAMQP1Session; cons
begin
  ShowMessage('#link-open: ' + aLink.Name);
end;
```

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.CreateReceiverLink('MySession', 'MyReceiverLink');
    procedure procedure TfrmClientAMQP1.AMQP1AMQPLinkOpen(Sender: TObject; const aSession: TsgcAMQP1Session; cons
    begin
      ShowMessage('#link-open: ' + aLink.Name);
    end;
```

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.

```
    procedure OnAMQPMessageEvent(Sender: TObject; const aSession:
        TsgcAMQP1Session; const aLink: TsgcAMQP1ReceiverLink; const aMessage:
        TsgcAMQP1Message; var DeliveryState: TsgcAMQP1MessageDeliveryState);
    begin
      ShowMessage(aMessage.ApplicationData.AMQPValue.Value);
    end;
```

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 := TsgcWSPClient_AMQP1.Create(nil);
// Creating WebSocket client
oClient := TsgcWebSocketClient.Create(nil);
// Setting WebSocket specifications
oClient.Specifications.RFC6455 := False;
// Setting WebSocket client properties
oClient.Host := '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 := TsgcWSPClient_AMQP1.Create(nil);
// Setting AMQP authentication options
oAMQP.AMQPOptions.Authentication.AuthType := amqp1authSASLPlain;
oAMQP.AMQPOptions.Authentication.Username := 'sgc';
oAMQP.AMQPOptions.Authentication.Password := 'sgc';
// Creating WebSocket client
oClient := TsgcWebSocketClient.Create(nil);
// Setting WebSocket specifications
oClient.Specifications.RFC6455 := False;
// Setting WebSocket client properties
oClient.Host := '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.

```
procedure Close(const aCondition, aDescription: string);
var
  oOptions: TsgcAMQP1MethodOptions_Close;
begin
  oOptions := TsgcAMQP1MethodOptions_Close.Create;
  Try
    oOptions.ErrorCondition := aCondition;
        oOptions.ErrorDescription := aDescription;
    oOptions.Await := True;
    AMQP1.Close(oOptions);
  Finally
    oOptions.Free;
  End;
end;
```

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**.

```
procedure OnAMQP1SASLAuthentication(Sender: TObject;
  aCode: TsgcAMQP1SaslCode; const aDescription: string; var Handled: Boolean);
begin
  ShowMessage('#sasl-authentication: ' + aDescription);
end;
```

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
oClient := TsgcWebSocketClient.Create(nil);
oClient.Specifications.RFC6455 := False;
oClient.Host := 'esegece.servicebus.windows.net';
oClient.Port := 5671;
oClient.TLS := True;
// ... create amqp1 protocol client
oAMQP1 := TsgcWSClient_AMQP1.Create(nil);
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
oClient := TsgcWebSocketClient.Create(nil);
oClient.Specifications.RFC6455 := False;
oClient.Host := 'esegece.servicebus.windows.net';
oClient.Port := 5671;
oClient.TLS := True;
// ... create AMQP1 protocol client
oAMQP1 := TsgcWSClient_AMQP1.Create(nil);
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
oClient := TsgcWebSocketClient.Create(nil);
oClient.Specifications.RFC6455 := False;
oClient.Host := 'esegece.servicebus.windows.net';
oClient.Port := 5671;
oClient.TLS := True;
// ... create AMQP1 protocol client
oAMQP1 := TsgcWSClient_AMQP1.Create(nil);
oAMQP1.Specifications.RFC6455 := False;
oAMQP1.AMQPOptions.Authentication.AuthType := amqp1authSASLAnonymous;
oAMQP1.Client := oClient;
// ... connect and publish JWT through CBS
oClient.Active := True;
// wait the client is connected
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');
procedure OnAMQPSessionOpen(Sender: TObject; const aSession: TsgcAMQP1Session; const aBegin: TsgcAMQP1FrameBegin)
begin
  ShowMessage('#session-open: ' + aSession.Id);
end;
```

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.

```
procedure OpenSession(const aSession: string);
var
  oOptions: TsgcAMQP1MethodOptions_SessionOpen;
begin
  oOptions := TsgcAMQP1MethodOptions_SessionOpen.Create;
```

```
  Try
    oOptions.Await := True;
    AMQP1.CreateSession(aSession, oOptions);
  Finally
    oOptions.Free;
  End;
end;
```

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');
procedure OnAMQPSessionCloseEvent(Sender: TObject; const aSession: TsgcAMQP1Session; const aEnd: TsgcAMQP1FrameEr
begin
  ShowMessage('#session-close: ' + aSession.Id + ' [' + IntToStr(aSession.Channel) + ']' + ' reason: ' + aEnd.Err
end;
```

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.

```
procedure CloseSession(const aSession: string);
var
  oOptions: TsgcAMQP1MethodOptions_SessionClose;
begin
  oOptions := TsgcAMQP1MethodOptions_SessionClose.Create;
  Try
    oOptions.Await := True;
    AMQP1.CloseSession(aSession, oOptions);
  Finally
    oOptions.Free;
  End;
end;
```

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');
procedure procedure TfrmClientAMQP1.AMQP1AMQPLinkOpen(Sender: TObject; const aSession: TsgcAMQP1Session; const aL
begin
  ShowMessage('#link-open: ' + aLink.Name);
end;
```

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.

```
procedure CreateSenderLink(const aSession, aSender: string);
var
  oOptions: TsgcAMQP1MethodOptions_CreateSenderLink;
begin
  oOptions := TsgcAMQP1MethodOptions_CreateSenderLink.Create;
  Try
    oOptions.Await := True;
    AMQP1.CreateSenderLink(aSession, aSender, '', oOptions);
  Finally
    oOptions.Free;
  End;
end;
```

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.

```
oMessageOptions := TsgcAMQP1MessageOptions.Create;
Try
  oMessageOptions.Settled := True;
  oAMQP1.SendMessage('MySession', 'MySenderLink', 'MyMessage', 'message-id', oMessageOptions);
Finally
  oMessageOptions.Free;
End;
```

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');
procedure OnAMQPLinkCloseEvent(Sender: TObject; const aSession: TsgcAMQP1Session; const aLink: TsgcAMQP1Link; cor
begin
  ShowMessage('#link-close: ' + aLink.Name);
end;
```

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.

```
procedure CloseSenderLink(const aSession, aSenderLink: string);
var
  oOptions: TsgcAMQP1MethodOptions_CloseLink;
begin
  oOptions := TsgcAMQP1MethodOptions_CloseLink.Create;
  Try
    oOptions.Await := True;
    AMQP1.CloseLink(aSession, aSenderLink, oOptions);
  Finally
    oOptions.Free;
  End;
end;
```

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');
procedure procedure TfrmClientAMQP1.AMQP1AMQPLinkOpen(Sender: TObject; const aSession: TsgcAMQP1Session; const aL
begin
  ShowMessage('#link-open: ' + aLink.Name);
end;
```

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.

```
procedure CreateReceiverLink(const aSession, aReceiver: string);
var
  oOptions: TsgcAMQP1MethodOptions_CreateReceiverLink;
begin
  oOptions := TsgcAMQP1MethodOptions_CreateReceiverLink.Create;
  Try
    oOptions.Await := True;
    AMQP1.CreateReceiverLink(aSession, aReceiver, '', oOptions);
  Finally
    oOptions.Free;
  End;
end;
```

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');
procedure OnAMQPLinkCloseEvent(Sender: TObject; const aSession: TsgcAMQP1Session; const aLink: TsgcAMQP1Link; cor
begin
  ShowMessage('#link-close: ' + aLink.Name);
end;
```

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.

```
procedure CloseReceiverLink(const aSession, aReceiverLink: string);
var
  oOptions: TsgcAMQP1MethodOptions_CloseLink;
begin
  oOptions := TsgcAMQP1MethodOptions_CloseLink.Create;
  Try
    oOptions.Await := True;
    AMQP1.CloseLink(aSession, aReceiverLink, oOptions);
  Finally
    oOptions.Free;
  End;
end;
```

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.

```
procedure SendMessageAwait(const aSession, aSenderLink, aText: string);
var
  oOptions := TsgcAMQP1MethodOptions_SendMessageAck.Create;
begin
  Try
    oOptions.Settled := False;
    oOptions.Await := True;
    AMQP1.SendMessage(aSession, aSenderLink, aText, 'messsage-id', oOptions);
  Finally
    oOptions.Free;
  End;
end;
```

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.

```
procedure OnAMQPMessageSentAck(Sender: TObject;
const aSession: TsgcAMQP1Session; const aLink: TsgcAMQP1SenderLink;
const aMessageId: string; const aDeliveryState: TsgcAMQP1FrameDeliveryStates;
const aDisposition: TsgcAMQP1FrameDisposition);
var
  vMessageId: string;
begin
  vMessageId := aMessageId;
  case aDeliveryState of
    amqp1fdtsAccepted:
      ShowMessage('#msg-accepted: ' + vMessageId);
    amqp1fdtsRejected:
      ShowMessage('#msg-rejected: ' + vMessageId + ' ' + TsgcAMQP1FrameRejected
        (aDisposition.State).Error.Condition + ' ' + TsgcAMQP1FrameRejected
        (aDisposition.State).Error.Description);
    amqp1fdtsReleased:
      ShowMessage('#msg-released: ' + vMessageId);
    amqp1fdtsModified:
      ShowMessage('#msg-modified: ' + vMessageId + ' ' + TsgcAMQP1FrameModified
        (aDisposition.State).MessageAnnotations);
    amqp1fdtsReceived:
      ShowMessage('#msg-received: ' + vMessageId);
  end;
end;
```

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.

```
procedure OnAMQPMessage(Sender: TObject;
const aSession: TsgcAMQP1Session; const aLink: TsgcAMQP1ReceiverLink;
const aMessage: TsgcAMQP1Message;
var DeliveryState: TsgcAMQP1MessageDeliveryState);
begin
  if aMessage.ApplicationData.AMQPValue.Value = 'xxx' then
  begin
    DeliveryState.State := amqp1mdtsRejected;
    DeliveryState.Rejected.Error.Condition := 'amqp-error-processing';
    DeliveryState.Rejected.Error.Description := 'Value received was not the expected.';
  end
  else
    DeliveryState.State := amqp1mdtsAccepted;
end;
```

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 AppRTC

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.

appr.tc is a WebRTC demo application developed by Google and Mozilla, it enables both browsers to “talk” to each other using the WebRTC API.

Components

TsgcWSPServer_AppRTC: Server Protocol AppRTC VCL Component.

TsgcWSPServer_AppRTC

This is the Server Protocol AppRTC Component.

Introduction

This is the Server Protocol AppRTC Component. You need to drop this component in the form and select a TsgcWebSocketServer Component using Server Property.

Parameters

- **IceServers:** here you can configure turn/stun servers for WebRTC connections.
- **RoomLink:** URL base to access room. Example: https://mydemo.com/r/
- **WebSocketURL:** URL to WebSocket server. Example: wss://mydemo.com

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.

IceServers Configuration

If you are running your STUN/TURN server in the following IP Address: 51.122.4.88 and is listening port 3478. User to connect is "apprtc" and credential is "secret". Configure the IceServers as follows:

```
{
        "lifetimeDuration": "86400s",
        "iceServers": [{
                "urls": "stun:51.122.4.88:3478",
                "username": "apprtc",
                "credential": "secret"
        }, {
                "urls": "turn:51.122.4.88:3478",
                "username": "apprtc",
                "credential": "secret"
        }],
        "blockStatus": "NOT_BLOCKED",
        "iceTransportPolicy": "all"
}
```

Reference

- Properties — 5 properties available on the component.
- Methods — 0 public methods.
- Events — 11 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

```
procedure OnMessageEvent(Connection: TsgcWSConnection; const Text: string);
begin
  ShowMessage(Text);
end;

oClient := TsgcWebSocketClient.Create(nil);
oClient.Host := '127.0.0.1';
oClient.Port := 80;
oClientWAMP := TsgcWSPClient_WAMP.Create(nil);
oClientWAMP.Client := oClient.
oClientWAMP.OnMessage := OnMessageEvent;
oClient.Active := True;

// Subscribe to topic after successful connect
oClient.Subscribe('myTopic');
```

WAMP Server

```
procedure OnSubscriptionEvent(Connection: TsgcWSConnection; const Subscription: string);
begin
  ShowMessage('Subscribed: ' + Subscription);
end;

oServer := TsgcWebSocketServer.Create(nil);
oServer.Port := 80;
oServerWAMP := TsgcWSPServer_WAMP.Create(nil);
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 := TsgcWebSocketClient.Create(nil);
oClient.Host := '127.0.0.1';
oClient.Port := 80;
oClientWAMP := TsgcWSPClient_WAMP.Create(nil);
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

```
procedure OnServerCall(Connection: TsgcWSConnection; const CallId, ProcUri, Arguments: string);
begin
  if ProcUri = 'GetTime' then
    oServerWAMP.CallResult(CallId, FormatDateTime('yyyymmdd hh:nn:ss', Now))
  else
    oServer.WAMP.CallError(CallId, 'Unknown method');
end;
oServer := TsgcWebSocketServer.Create(nil);
oServer.Port := 80;
oServerWAMP := TsgcWSPServer_WAMP.Create(nil);
oServerWAMP.OnCall := OnServerCallEvent;
oServerWAMP.Server := oServer;
oServer.Active := True;
```

WAMP Client

```
procedure OnCallResultClient(Connection: TsgcWSConnection; CallId, Result: string);
begin
  ShowMessage(Result);
end;
procedure OnCallErrorClient(Connection: TsgcWSConnection; const Error: string);
begin
  ShowMessage(Error);
end;
oClient := TsgcWebSocketClient.Create(nil);
oClient.Host := '127.0.0.1';
oClient.Port := 80;
oClientWAMP := TsgcWSPClient_WAMP.Create(nil);
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

```
procedure OnServerCall(Connection: TsgcWSConnection; const CallId, ProcUri, Arguments: string);
var
  vNum: Integer;
begin
  if ProcUri = 'GetProgressiveTime' then
  begin
    vNum := StrToInt(Arguments);
    for i := 1 to vNum do
        begin
          if i = 20 then
        oServerWAMP.CallResult(CallId, FormatDateTime('yyyymmdd hh:nn:ss', Now))
          else
            oServerWAMP.CallProgressiveResult(CallId, FormatDateTime('yyyymmdd hh:nn:ss', Now));
        end
  end
  else
    oServer.WAMP.CallError(CallId, 'Unknown method');
end;


oServer := TsgcWebSocketServer.Create(nil);
oServer.Port := 80;
oServerWAMP := TsgcWSPServer_WAMP.Create(nil);
oServerWAMP.OnCall := OnServerCallEvent;
oServerWAMP.Server := oServer;
oServer.Active := True;
```

WAMP Client

```
procedure OnCallResultClient(Connection: TsgcWSConnection; CallId, Result: string);
begin
  ShowMessage(Result);
end;


procedure OnCallProgressResultClient(Connection: TsgcWSConnection; CallId, Result: string);
begin
  ShowMessage(Result);
end;


procedure OnCallErrorClient(Connection: TsgcWSConnection; const Error: string);
begin
  ShowMessage(Error);
end;


oClient := TsgcWebSocketClient.Create(nil);
oClient.Host := '127.0.0.1';
oClient.Port := 80;
oClientWAMP := TsgcWSPClient_WAMP.Create(nil);
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

```
procedure OnAfterOpenDataSet(DataSet: TDataSet);
begin
  DataSet.FieldByName('ID').ProviderFlags :=
    Dataset.FieldByName('ID').ProviderFlags + [pfInKey];
end;
```

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.

```
oServer := TsgcWebSocketServer.Create(nil);
oProtocolDataset := TsgcWSPServer_Dataset.Create(nil);
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.

```
oClient := TsgcWebSocketClient.Create(nil);
oProtocolDataset := TsgcWSPClient_Dataset.Create(nil);
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**.

```
oServer := TsgcWebSocketServer.Create(nil);
oProtocolDataset := TsgcWSPServer_Dataset.Create(nil);
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**.

```
oClient := TsgcWebSocketClient.Create(nil);
oProtocolDataset := TsgcWSPClient_Dataset.Create(nil);
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
oServer := TsgcWebSocketServer.Create(nil);
oServer_Files := TsgcWSPServer_Files.Create(nil);
oServer_Files.Server := oServer;
oServer.Host := '127.0.0.1';
oServer.Port := 8080;


// ... Create Client
oClient := TsgcWebSocketClient.Create(nil);
oClient.URL := 'ws://127.0.0.1:8080';


// ... Create Protocol
oClient_Files := TsgcWSPClient_Files.Create(nil);
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
procedure OnConnectEvent(Connection: TsgcWSConnection);
begin
  FGuid := Connection.Guid;
end;


// ... Create Server
oServer := TsgcWebSocketServer.Create(nil);
oServer_Files := TsgcWSPServer_Files.Create(nil);
oServer_Files.Server := oServer;
oServer_Files.OnConnect := OnConnectEvent;
oServer.Host := '127.0.0.1';
oServer.Port := 8080;


// ... Create Client
oClient := TsgcWebSocketClient.Create(nil);
oClient.URL := 'ws://127.0.0.1:8080';


// ... Create Protocol
oClient_Files := TsgcWSPClient_Files.Create(nil);
oClient_Files.Client := oClient;


// ... Start Server and Connect Clients
oServer.Active := True;
oClient.Connect();


// ... Send File to the client connected
oServer_Files.SendFile(FConnection.Guid, '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
oServer := TsgcWebSocketServer.Create(nil);
oServer_Files := TsgcWSPServer_Files.Create(nil);
oServer_Files.Server := oServer;
oServer.Host := '127.0.0.1';
oServer.Port := 8080;


// ... Create Client
oClient := TsgcWebSocketClient.Create(nil);
oClient.URL := 'ws://127.0.0.1:8080';


// ... Create Protocol
oClient_Files := TsgcWSPClient_Files.Create(nil);
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
FXCM	also known as Forex Capital Markets, is a retail broker for trading on the foreign
Bitfinex	Bitfinex is one of the world's largest and most advanced cryptocurrency trading Bitcoin, Ethereum, Ripple, EOS, Bitcoin Cash, Iota, NEO, Litecoin, Ethereum Clas
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 tradin
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
RTCMultiConnection	RTCMultiConnection is a WebRTC JavaScript library for peer-to-peer app video conferencing, file sharing, media streaming etc.)
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

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).
- **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.
    - **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)

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.

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.

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.
GetAveragePrice	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.
GetRollingWindowTicker	Symbol, Symbols, WindowSize	Rolling window price change statistics. Note: WindowSize default is 1d if not specified.
GetTradingDayTicker	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.
GetWalletAssetDividen-dRecord	Query asset dividend record.
GetWalletAssetDetail	Fetch details of assets supported on Binance.
GetWalletTradeFee	Fetch trade fee
WalletUserUniver-salTransfer	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).

(*) 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 TsgcWebSocketClient.
See below an example:

```
oClient := TsgcWebSocketClient.Create(nil);
oBinance := TsgcWSAPI_Binance.Create(nil);
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:

```
oClient := TsgcWebSocketClient.Create(nil);
oBinance := TsgcWSAPI_Binance.Create(nil);
oBinance.Client := oClient;
oBinance.SubscribeTicker('bnbbtc');

procedure OnMessage(Connection: TsgcWSConnection; const aText: string);
begin
// here you will receive the ticker updates
end;
```

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:

```
oBinance := TsgcWSAPI_Binance.Create(nil);
ShowMessage(oBinance.REST_API.GetPriceTicker('BNBBTC'));
```

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.

```
oBinance := TsgcWSAPI_Binance.Create(nil);
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

```
oBinance := TsgcWSAPI_Binance.Create(nil);
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

```
oBinance := TsgcWSAPI_Binance.Create(nil);
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**.

Binance | Withdraw

Binance allows you to use the Wallet API to submit a Withdraw request, only the following parameters are mandatory:

- Coin
- Address
- Amount

```
oBinance := TsgcWSAPI_Binance.Create(nil);
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).
- **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)

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.

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 invest-

	ment 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.

GetOpenInterestStatistics	Symbol, Period	
GetTopTraderAccountRatio	Symbol, Period	
GetTopTraderPositionRatio	Symbol, Period	
GetGlobalAccountRatio	Symbol, Period	
GetTakerVolume	Symbol, Period	
GetContinuousKLines	Pair, ContractType, Interval	Kline/candlestick bars for a specific contract type.
GetIndexPriceKLines	Pair, Interval	Kline/candlestick bars for the index price of a pair.
GetMarkPriceKLines	Symbol, Interval	Kline/candlestick bars for the mark price of a symbol.
GetPremiumIndexKLines	Symbol, Interval	Premium index kline bars of a symbol.
GetFundingInfo		Get funding rate info for all symbols.
GetPriceTickerV2	Symbol	Latest price for a symbol or symbols (V2).
GetIndexInfo	Symbol	Get index info.
GetAssetIndex	Symbol	Get asset index for multi-assets mode.
GetConstituents	Symbol	Get index constituents.
GetDeliveryPrice	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.
CancelAl-lOpenOrders	Symbol	
AutoCancelAl-lOpenOrders	Symbol, CountDownTimer	Cancel all open orders of the specified symbol at the end of the specified countdown.
QueryCurren-tOpenOrder	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.
GetAccountBal-ance		
GetAccountIn-formation		Get current account information.
ChangeInitial-Leverage	Symbol, Leverage	Change user's initial leverage of specific symbol market.
ChangeMargin-Type	Symbol, MarginType	
ModifyIsolated-PositionMargin	Symbol, Amount, Type	
GetPosition-MarginChange-History	Symbol	
GetPositionIn-formation	Symbol	
GetAccount-TradeList	Symbol	
GetIncomeHis-tory	Symbol	
GetNotional-LeverageBrack-et	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.

GetAccountBalanceV3		Get futures account balance (V3).
GetAccountInformationV3		Get current account information (V3).
GetPositionInformationV3	Symbol	Get current position information (V3).
GetCommissionRate	Symbol	Get user commission rate.
GetAccountConfig		Get current account configuration.
GetSymbolConfig	Symbol	Get symbol configuration.
GetOrderRateLimit		Get user's order rate limit.
GetApiTradingStatus	Symbol	Get API trading quantitative rules indicators.
ChangeMultiAssetsMode	MultiAssetsMargin	Change user's multi-assets mode. Multi-Assets Mode: true; Single-Asset Mode: false.
GetMultiAssetsMode		Get user's current multi-assets mode.
SetFeeBurn	FeeBurn	Change user's BNB fee burn status.
GetFeeBurn		Get user's BNB fee burn status.
CreateListenKey		Start a new user data stream. The stream will close after 60 minutes unless a keepalive is sent.
KeepAliveListenKey		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).

(*) 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.

```
procedure TfrmWebSocketClient.SOCKETIOHTTPConnectionSSL(Sender: TObject;
    aSSLHandler: TIdSSLIOHandlerSocketBase);
begin
  TIdSSLIOHandlerSocketOpenSSL(aSSLHandler).SSLOptions.Method := sslvTLSv1_2;
end;
```

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

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	Get rates for a single product by product ID, grouped in buckets.

	aGranularity: The timeframe each candle represents.	
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:

```
oClient := TsgcWebSocketClient.Create(nil);
oCoinbase := TsgcWSAPI_Coinbase.Create(nil);
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:

```
oClient := TsgcWebSocketClient.Create(nil);
oCoinbase := TsgcWSAPI_Coinbase.Create(nil);
oCoinbase.Client := oClient;
oCoinbase.SubscribeTicker('ETH-USD');

procedure OnCoinbaseMessage(Sender: TObject; aType, aRawMessage: string);
begin
// here you will receive the ticker updates
end;
```

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

```
oCoinbase := TsgcWSAPI_Coinbase.Create(nil);
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

```
oCoinbase := TsgcWSAPI_Coinbase.Create(nil);
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

```
oCoinbase := TsgcWSAPI_Coinbase.Create(nil);
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

```
oCoinbase := TsgcWSAPI_Coinbase.Create(nil);
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.

```
oCoinbase := TsgcWSAPI_Coinbase.Create(nil);
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
WSClient := TsgcWebSocketClient.Create(nil);
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
Signal := TsgcWSAPI_SignalRCore.Create(nil);
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', ['John', 'Hello All.'], 'id-000001');

procedure OnSignalRCoreCompletion(Sender: TObject; Completion: TSignalRCore_Completion);
begin
  if Completion.Error <> '' then
    ShowMessage('Something goes wrong.')
  else
    ShowMessage('Invocation Successful!');
end;
```

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', ['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');

procedure OnSignalRCoreStreamItem(Sender: TObject; StreamItem: TSignalRCore_StreamItem; var Cancel: Boolean);
begin
  DoLog('#stream item: ' + StreamItem.Item);
end;

procedure OnSignalRCoreCompletion(Sender: TObject; Completion: TSignalRCore_Completion);
begin
  if Completion.Error  '' then
    ShowMessage('Something goes wrong.')
  else
    ShowMessage('Invocation Successful!');
end;
```

Invocations

In order to perform a single invocation, the Caller follows the following basic flow:

```
procedure Invoke(const aTarget: String; const aArguments: Array of Const; const aInvocationId: String = '');
procedure InvokeStream(const aTarget: String; const aArguments: Array of Const; const aInvocationId: String);
```

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');

procedure OnSignalRCoreStreamItem(Sender: TObject; StreamItem: TSignalRCore_StreamItem; var Cancel: Boolean);
begin
  if StreamItem.InvocationId = 'id-000002' then
    DoLog('#stream item: ' + StreamItem.Item);
end;

procedure OnSignalRCoreCompletion(Sender: TObject; Completion: TSignalRCore_Completion);
begin
  if StreamItem.InvocationId = 'id-000002' then
  begin
    if Completion.Error  '' then
      ShowMessage('Something goes wrong.')
    else
      ShowMessage('Invocation Successful!');
  end;
end;
```

You can call a single invocation and wait for completion.

```
function InvokeAndWait(const aTarget: String; aArguments: Array of Const; aInvocationId: String; out Completion:
  const aTimeout: Integer = 10000): Boolean;
function InvokeStreamAndWait(const aTarget: String; const aArguments: Array of Const; const aInvocationId: String
  out Completion: TSignalRCore_Completion; const aTimeout: Integer = 10000): Boolean;
```

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.

```
var
  oCompletion: TSignalRCore_Completion;
begin
  if SignalRCore.InvokeStreamAndWait('Counter', [10, 500], 'id-000002', oCompletion) then
    DoLog('#invoke stream ok: ' + oCompletion.Result)
  else
    DoLog('#invocke stream error: ' + oCompletion.Error);


procedure OnSignalRCoreStreamItem(Sender: TObject; StreamItem: TSignalRCore_StreamItem; var Cancel: Boolean);
begin
  if StreamItem.InvocationId = 'id-000002' then
    DoLog('#stream item: ' + StreamItem.Item);
end;
```

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.

```
procedure OnSignalRCoreStreamItem(Sender: TObject; StreamItem: SignalRCore_StreamItem; var Cancel: Boolean);
begin
  if StreamItem.InvocationId = 'id-000002' then
    Cancel := True;
end;
```

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.

```
procedure OnSignalRCoreInvocation(Sender: TObject; Invocation: TSignalRCore_Invocation);
begin
  if Invocation.Target = 'SendMessage' then
    ... your code here ...
end;
// 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.
procedure OnSignalRCoreClose(Sender: TObject; Close: TSignalRCore_Close);
begin
  DoLog('#closed: ' + Close.Error);
end;
```

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.

```
procedure OnSignalRCoreKeepAlive(Sender: TObject);
begin
  DoLog('#keepalive');
end;
```

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":["Delphi 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.

```
oSignalR := TsgcWSAPI_Signal.Create(nil);
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.

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

```
oClient := TsgcWebSocketClient.Create(nil);
oKraken := TsgcWSAPI_Kraken.Create(nil);
oKraken.Client := oClient;
oClient.Active := True;
```

How to Connect to Private WebSocket Server

```
oClient := TsgcWebSocketClient.Create(nil);
oKraken := TsgcWSAPI_Kraken.Create(nil);
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

```
oClient := TsgcWebSocketClient.Create(nil);
oKraken := TsgcWSAPI_Kraken.Create(nil);
```

```
oKraken.Client := oClient;
ShowMessage(oKraken.GetTicker(['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

```
oClient := TsgcWebSocketClient.Create(nil);
oKraken := TsgcWSAPI_Kraken.Create(nil);
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(['XBT/USD']);
```

If subscription is successful, **OnKrakenSubscribed** event will be called:

```
procedure OnKrakenSubscribed(Sender: TObject;ChannelId: Integer; Pair, Subscription, ChannelName: string;
  ReqID:Integer);
begin
  DoLog('#subscribed: ' + Subscription + ' ' + Pair + ' ' + ChannelName);
end;
```

UnSubscribe calling UnSubscribeTicker method:

```
UnSubscribeTicker(['XBT/USD']);
```

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

```
procedure OnKrakenUnSubscribed(Sender: TObject; ChannelId: Integer; Pair, Subscription: string;
  ReqID: Integer);
begin
  DoLog('#unsubscribed: ' + Subscription + ' ' + Pair);
end;
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenSubscriptionError event will be called.

```
procedure OnKrakenSubscriptionError(Sender: TObject; ErrorMessage, Pair, Subscription: string;
  ReqID: Integer);
begin
  DoLog('#subscription error: ' + ErrorMessage);
end;
```

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(['XBT/USD'], kin1min);
```

If subscription is successful, OnKrakenSubscribed event will be called:

```
procedure OnKrakenSubscribed(Sender: TObject;ChannelId: Integer; Pair, Subscription, ChannelName: string;
  ReqID:Integer);
begin
  DoLog('#subscribed: ' + Subscription + ' ' + Pair + ' ' + ChannelName);
end;
```

UnSubscribe calling UnSubscribeOHLC method:

```
UnSubscribeOHLC(['XBT/USD'], kin1min);
```

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

```
procedure OnKrakenUnSubscribed(Sender: TObject; ChannelId: Integer; Pair, Subscription: string;
  ReqID: Integer);
begin
  DoLog('#unsubscribed: ' + Subscription + ' ' + Pair);
end;
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenSubscriptionError event will be called.

```
procedure OnKrakenSubscriptionError(Sender: TObject; ErrorMessage, Pair, Subscription: string;
  ReqID: Integer);
begin
  DoLog('#subscription error: ' + ErrorMessage);
end;
```

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(['XBT/USD']);
```

If subscription is successful, OnKrakenSubscribed event will be called:

```
procedure OnKrakenSubscribed(Sender: TObject;ChannelId: Integer; Pair, Subscription, ChannelName:
  string; ReqID:Integer);
begin
  DoLog('#subscribed: ' + Subscription + ' ' + Pair + ' ' + ChannelName);
end;
```

UnSubscribe calling UnSubscribeTrade method:

```
UnSubscribeTrade(['XBT/USD']);
```

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

```
procedure OnrakenUnSubscribed(Sender: TObject; ChannelId: Integer; Pair, Subscription: string;
  ReqID: Integer);
begin
  DoLog('#unsubscribed: ' + Subscription + ' ' + Pair);
end;
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenSubscriptionError event will be called.

```
procedure OnKrakenSubscriptionError(Sender: TObject; ErrorMessage, Pair, Subscription: string;
  ReqID: Integer);
begin
  DoLog('#subscription error: ' + ErrorMessage);
end;
```

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(['XBT/USD'], kde10);
```

If subscription is successful, OnKrakenSubscribed event will be called:

```
procedure OnKrakenSubscribed(Sender: TObject;ChannelId: Integer; Pair, Subscription, ChannelName: string;
  ReqID:Integer);
begin
  DoLog('#subscribed: ' + Subscription + ' ' + Pair + ' ' + ChannelName);
end;
```

UnSubscribe calling UnSubscribeBook method:

```
UnSubscribeBook(['XBT/USD'], kde10);
```

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

```
procedure OnKrakenUnSubscribed(Sender: TObject; ChannelId: Integer; Pair, Subscription: string;
  ReqID: Integer);
begin
  DoLog('#unsubscribed: ' + Subscription + ' ' + Pair);
end;
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenSubscriptionError event will be called.

```
procedure OnKrakenSubscriptionError(Sender: TObject; ErrorMessage, Pair, Subscription: string; ReqID: Integer);
begin
  DoLog('#subscription error: ' + ErrorMessage);
end;
```

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(['XBT/USD']);
```

If subscription is successful, OnKrakenSubscribed event will be called:

```
procedure OnKrakenSubscribed(Sender: TObject;ChannelId: Integer; Pair, Subscription, ChannelName: string;
  ReqID:Integer);
begin
  DoLog('#subscribed: ' + Subscription + ' ' + Pair + ' ' + ChannelName);
end;
```

UnSubscribe calling UnSubscribeSpread method:

```
UnSubscribeSpread(['XBT/USD']);
```

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

```
procedure OnrakenUnSubscribed(Sender: TObject; ChannelId: Integer; Pair, Subscription: string;
  ReqID: Integer);
begin
  DoLog('#unsubscribed: ' + Subscription + ' ' + Pair);
end;
```

If there is an error while trying to subscribe / unsubscribe, OnKrakenSubscriptionError event will be called.

```
procedure OnKrakenSubscriptionError(Sender: TObject; ErrorMessage, Pair, Subscription: string;
  ReqID: Integer);
begin
  DoLog('#subscription error: ' + ErrorMessage);
end;
```

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(['XBT/USD']);

UnSubscribeAll(['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

```
oKrakenOrder := TsgcWSKrakenOrder.Create;
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(['XBTUSD']);
```

GetTicker

Returns ticker information

```
Kraken.REST_API.GetTicker(['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
```

```
oKrakenOrder := TsgcHTTPKrakenOrder.Create;
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 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

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(['PI_XBTUSD']);
```

If subscription is successful, **OnKrakenFuturesSubscribed** event will be called:

```
procedure OnKrakenFuturesSubscribed(Sender: TObject; Feed, ProductId: string);
begin
  DoLog('#subscribed: ' + Feed + ' ' + ProductId);
end;
```

UnSubscribe calling **UnSubscribeTicker** method:

```
UnSubscribeTicker(['PI_XBTUSD']);
```

If unsubscription is successful, **OnKrakenFuturesUnSubscribed** event will be called:

```
procedure OnKrakenFuturesUnSubscribed(Sender: TObject; Feed, ProductId: string);
begin
  DoLog('#unsubscribed: ' + Feed + ' ' + ProductId);
end;
```

If there is an error while trying to subscribe / unsubscribe, **OnKrakenFuturesError** event will be called.

```
procedure OnKrakenFuturesError(Sender: TObject; Error: string);
begin
  DoLog('#error: ' + Error);
end;
```

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(['PI_XBTUSD']);
```

If subscription is successful, **OnKrakenFuturesSubscribed** event will be called:

```
procedure OnKrakenFuturesSubscribed(Sender: TObject; Feed, ProductId: string);
begin
  DoLog('#subscribed: ' + Feed + ' ' + ProductId);
end;
```

UnSubscribe calling **UnSubscribeTrade** method:

```
UnSubscribeTrade(['PI_XBTUSD']);
```

If unsubscription is successful, **OnKrakenFuturesUnSubscribed** event will be called:

```
procedure OnrakenFuturesUnSubscribed(Sender: TObject; Feed, ProductId: string);
begin
  DoLog('#unsubscribed: ' + Feed + ' ' + ProductId);
end;
```

If there is an error while trying to subscribe / unsubscribe, **OnKrakenFuturesError** event will be called.

```
procedure OnKrakenFuturesSubscriptionError(Sender: TObject; Error: string);
begin
  DoLog('#error: ' + Error);
end;
```

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(['PI_XBTUSD']);
```

If subscription is successful, **OnKrakenFuturesSubscribed** event will be called:

```
procedure OnKrakenFuturesSubscribed(Sender: TObject; Feed, ProductId: string);
begin
  DoLog('#subscribed: ' + Feed + ' ' + ProductId);
end;
```

UnSubscribe calling **UnSubscribeBook** method:

```
UnSubscribeBook(['PI_XBTUSD']);
```

If unsubscription is successful, **OnKrakenFuturesUnSubscribed** event will be called:

```
procedure OnKrakenFuturesUnSubscribed(Sender: TObject; Feed, ProductId: string);
begin
  DoLog('#unsubscribed: ' + Feed + ' ' + ProductId);
end;
```

If there is an error while trying to subscribe / unsubscribe, **OnKrakenFuturesError** event will be called.

```
procedure OnKrakenFuturesError(Sender: TObject; Error: string);
begin
  DoLog('#error: ' + Error);
end;
```

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(['PI_XBTUSD']);
```

If subscription is successful, **OnKrakenFuturesSubscribed** event will be called:

```
procedure OnKrakenFuturesSubscribed(Sender: TObject; Feed, ProductId: string);
begin
  DoLog('#subscribed: ' + Feed + ' ' + ProductId);
end;
```

UnSubscribe calling **UnSubscribeTickerLite** method:

```
UnSubscribeTickerLite(['PI_XBTUSD']);
```

If unsubscription is successful, **OnKrakenFuturesUnSubscribed** event will be called:

```
procedure OnKrakenFuturesUnSubscribed(Sender: TObject; Feed, ProductId: string);
begin
  DoLog('#unsubscribed: ' + Feed + ' ' + ProductId);
end;
```

If there is an error while trying to subscribe / unsubscribe, **OnKrakenFuturesError** event will be called.

```
procedure OnKrakenFuturesError(Sender: TObject; Error: string);
begin
  DoLog('#error: ' + Error);
end;
```

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.

```
oOrder := TsgcHTTPKrakenFuturesOrder.Create;
Try
  oOrder.Side := kosfBuy;
  oOrder.Symbol := 'PI_XBTUSD';
  oOrder.OrderType := kotfMKT;
  oOrder.Size := 1;
  KrakenFutures.REST_API.SendOrder(oOrder);
Finally
```

```
  oOrder.Free;
End;
```

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".

```
Delphi
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".

```
Delphi
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:

```
procedure OnPusherAuthenticationEvent(Sender: TObject;
  AuthRequest: TsgcWSPusherRequestAuthentication;
  AuthResponse: TsgcWSPusherResponseAuthentication);
begin
  AuthResponse.SharedSecret := 'your-shared-secret-key';
end;
```

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.

```
procedure PUSHERPusherMemberAdded(Sender: TObject;
  Channel, UserId, UserInfo: string);
begin
  Log('Member joined: ' + UserId + ' on ' + Channel);
end;
```

OnPusherMemberRemoved

Raised when a member unsubscribes from a presence channel. Provides the channel name, user ID, and user info of the member that left.

```
procedure PUSHERPusherMemberRemoved(Sender: TObject;
  Channel, UserId, UserInfo: string);
begin
  Log('Member left: ' + UserId + ' on ' + Channel);
end;
```

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.

```
procedure PUSHERPusherSubscriptionCount(Sender: TObject;
  Channel: string; SubscriptionCount: Integer);
begin
  Log(Channel + ' has ' + IntToStr(SubscriptionCount) + ' subscribers');
end;
```

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.

```
procedure PUSHERPusherCacheMiss(Sender: TObject; Channel: string);
begin
  Log('Cache miss on: ' + Channel);
end;
```

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');
```

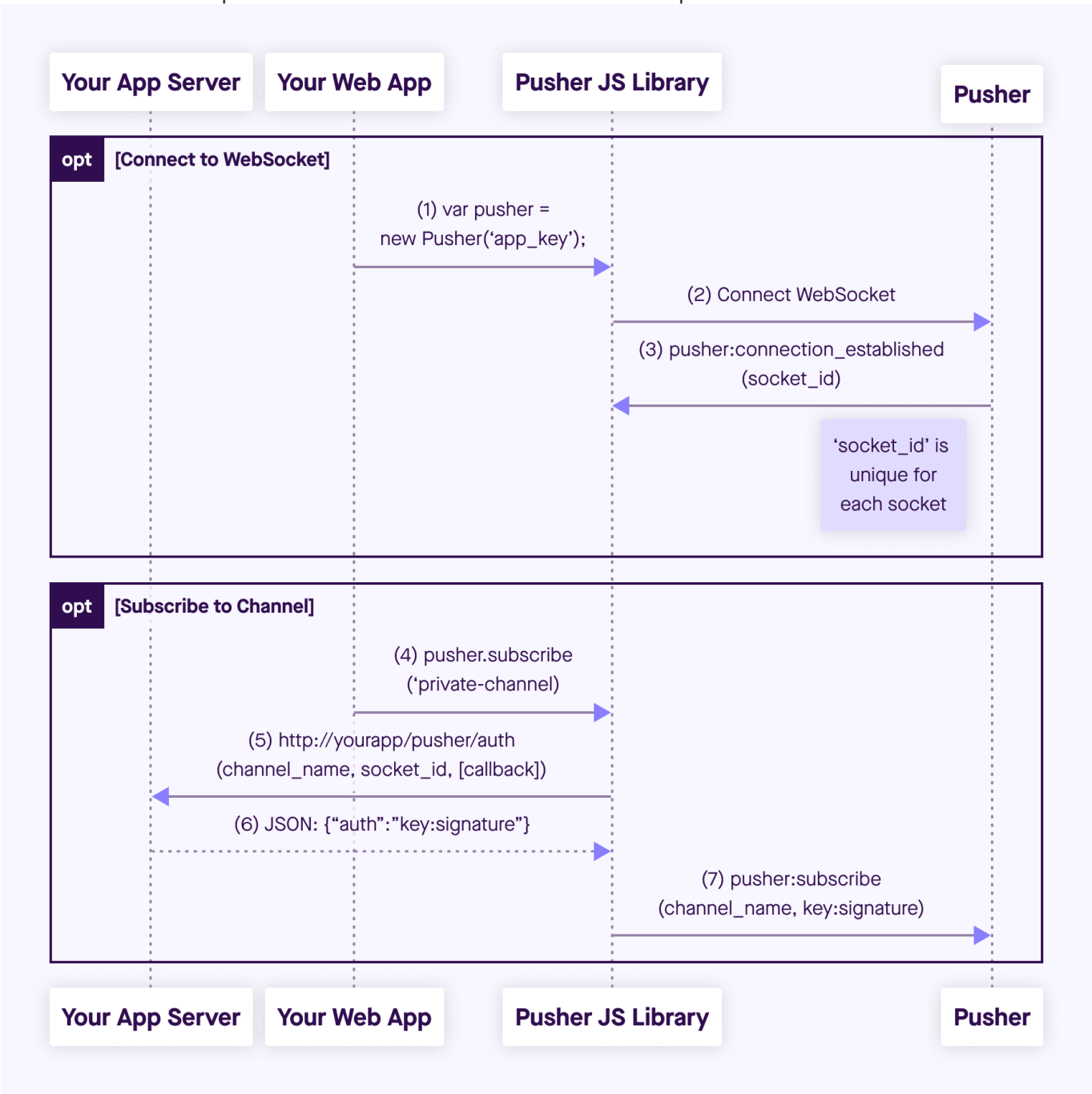
TerminateUserConnections

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:

```
oClient := TsgcWebSocketClient.Create(nil);
oPusher := TsgcWSAPI_Pusher.Create(nil);
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;

procedure OnPusherAuthenticationEvent(Sender: TObject; AuthRequest: TsgcWSPusherRequestAuthentication;
  AuthResponse: TsgcWSPusherResponseAuthentication);
begin
  // if the authentication request is successful return the signature
  if CustomAuthentication(AuthRequest.Channel, AuthRequest.SocketID) then
    AuthResponse.Signature := GetCustomAuthenticationSignature;
end;
```

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

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:

```
oClient := TsgcWebSocketClient.Create(nil);
oBitmex := TsgcWSAPI_Bitmex.Create(nil);
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:

```
oClient := TsgcWebSocketClient.Create(nil);
oBitmex := TsgcWSAPI_Bitmex.Create(nil);
oBitmex.Client := oClient;
oBitmex.Subscribe(btmTrade, 'XBTUSD');
procedure OnBitmexMessage(Sender: TObject; const aTopic: TwsBitmexTopics; const aMessage: string);
begin
// here you will receive the trade updates
end;
```

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.

If the server sends any information, this can be handled using **OnBitfinexInfoMessage** 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:

CS_DEC_S = 8; // Enable all decimal as strings.
CS_TIME_S = 32; // Enable all times as date strings.
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.

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.

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:

```
oClient := TsgcWebSocketClient.Create(nil);
oKucoin := TsgcWSAPI_Kucoin.Create(nil);
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:

```
oClient := TsgcWebSocketClient.Create(nil);
oKucoin := TsgcWSAPI_Kucoin.Create(nil);
oKucoin.Client := oClient;
oKucoin.SubscribeSymbolTicker('BTC-USDT');
procedure OnMessage(Connection: TsgcWSConnection; const aText: string);
begin
// here you will receive the ticker updates
end;
```

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

```
oKucoin := TsgcWSAPI_Kucoin.Create(nil);
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.

```
oKucoin := TsgcWSAPI_Kucoin.Create(nil);
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

```
oKucoin := TsgcWSAPI_Kucoin.Create(nil);
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

```
oKucoin := TsgcWSAPI_Kucoin.Create(nil);
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.

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 returned 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 market. 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 orders.
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 Order 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:

```
oClient := TsgcWebSocketClient.Create(nil);
oKucoin := TsgcWSAPI_Kucoin_Futures.Create(nil);
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:

```
oClient := TsgcWebSocketClient.Create(nil);
oKucoin := TsgcWSAPI_Kucoin_Futures.Create(nil);
oKucoin.Client := oClient;
oKucoin.SubscribeSymbolTickerV2('XBTUSDM');
procedure OnMessage(Connection: TsgcWSConnection; const aText: string);
begin
// here you will receive the ticker updates
end;
```

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

```
oKucoin := TsgcWSAPI_Kucoin_Futures.Create(nil);
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.

```
oKucoin := TsgcWSAPI_Kucoin_Futures.Create(nil);
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

```
oKucoin := TsgcWSAPI_Kucoin_Futures.Create(nil);
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

```
oKucoin := TsgcWSAPI_Kucoin_Futures.Create(nil);
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

CancelS-martTrade	**aId:** id of the trade	Cancel a Smart Trade by the Id of the Trade
CloseBy-MarketS-martTrade	**aId:** id of the trade	
EditSmart-Trade	**aId:** id of the trade	Edit an existing Smart Trade
ForceStartS-martTrade	**aId:** id of the trade	Force start a Smart Trade
AddFundsS-martTrade	**aId:** id of the trade	Add funds to a Smart Trade
GetSmart-Trade-Trades	**aId:** id of the trade	Get trades of a Smart Trade

DCA Bot

Method	Arguments	Description
CreateD-CABot		Create a new DCA Bot
GetD-CABot	**aId:** id of the bot	Get a DCA Bot by Id
GetD-CABots		Get all DCA Bots
EnableD-CABot	**aId:** id of the bot	Enable a DCA Bot
DisableD-CABot	**aId:** id of the bot	Disable a DCA Bot
DeleteD-CABot	**aId:** id of the bot	Delete a DCA Bot
CancelD-CABot	**aId:** id of the bot	Cancel a DCA Bot
GetD-CABot-Stats		Get DCA Bot statistics
GetAvail-ableStrat-egyList		Get available strategy list
GetBlack-listPairs		Get blacklist pairs
AddBlack-listPairs		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).

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oOKX := TsgcWSAPI_OKX.Create(nil);
oOKX.Client := oClient;
oOKX.OKX.ApiKey := 'alsdjk23kandfnasbdfdkjhsdf';
oOKX.OKX.ApiSecret := 'aldskjfk3jkadknfajndsjfj23j';
oOKX.OKX.Passphrase := 'secret_passphrase';
oClient.Active := True;
procedure OnOKXConnect(Sender: TObject; aMessage, aCode, aRawMessage: string);
begin
  DoLog('#OKX Connected');
end;
procedure OnOKXError(Sender: TObject; aCode, aMessage, aRawMessage: string);
begin
  DoLog('#error: ' + aMessage);
end;
```

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oOKX := TsgcWSAPI_OKX.Create(nil);
oOKX.Client := oClient;
oOKX.OKX.ApiKey := 'alsdjk23kandfnasbdfdkjhsdf';
oOKX.OKX.ApiSecret := 'aldskjfk3jkadknfajndsjfj23j';
oOKX.OKX.Passphrase := 'secret_passphrase';
oClient.Active := True;
procedure OnOKXConnect(Sender: TObject; aMessage, aCode, aRawMessage: string);
begin
  oOKX.SubscribeInstruments(okxitFutures);
end;
```

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oOKX := TsgcWSAPI_OKX.Create(nil);
oOKX.Client := oClient;
oOKX.OKX.ApiKey := 'alsdjk23kandfnasbdfdkjhsdf';
oOKX.OKX.ApiSecret := 'aldskjfk3jkadknfajndsjfj23j';
oOKX.OKX.Passphrase := 'secret_passphrase';
oClient.Active := True;
procedure OnOKXConnect(Sender: TObject; aMessage, aCode, aRawMessage: string);
begin
  oOKX.SubscribeOrders(okxitFutures, 'BTC-USD', 'BTC-USD-200329');
end;
```

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`

Parameter	Type	Required	Description
> 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.
> 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).

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oXTB := TsgcWSAPI_XTB.Create(nil);
oXTB.Client := oClient;
oXTB.XTB.User := 'user_0001';
oXTB.XTB.Password := 'secret_0001';
oClient.Active := True;
procedure OnXTBConnect(Sender: TObject; const aStreamSessionId: string);
begin
  DoLog('#XTB Connected');
end;
procedure OnXTBError(Sender: TObject; aCode, aDescription, aRawMessage: string);
begin
  DoLog('#error: ' + aDescription);
end;
```

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).

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oBybit := TsgcWSAPI_Bybit.Create(nil);
oBybit.Client := oClient;
oBybit.Bybit.ApiKey := 'alsdjk23kandfnasbdfdkjhsdf';
oBybit.Bybit.ApiSecret := 'aldskjfk3jkadknfajndsjfj23j';
oBybit.BybitClient := bybSpot;
oClient.Active := True;
procedure OnConnect(Connection: TsgcWSConnection);
begin
  DoLog('#Bybit Connected');
end;
```

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oBybit := TsgcWSAPI_Bybit.Create(nil);
oBybit.Client := oClient;
oBybit.Bybit.ApiKey := 'alsdjk23kandfnasbdfdkjhsdf';
oBybit.Bybit.ApiSecret := 'aldskjfk3jkadknfajndsjfj23j';
oBybit.BybitClient := bybSpot;
oClient.Active := True;
procedure OnBybitAuthentication(Sender: TObject; aSuccess: Boolean; const aError, aRawMessage: string)
begin
  if aSuccess then
  begin
    oClient.SubscribeOrderBook('BTCUSDT');
    oClient.SubscribeTrade('BTCUSDT');
  end;
end;
```

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oBybit := TsgcWSAPI_Bybit.Create(nil);
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.

```
oForex := TsgcWSAPI_Forex.Create(nil);
oForex.Credentials.UserName := '<username>';
oForex.Credentials.Password := '<password>';
oForex.Credentials.AppKey  := '<appkey>';
oForex.OnForexPriceTick := ForexPriceTick;
oForex.Connect;
oForex.WatchMarket(401484830); // EUR/USD market id
// ... receive price ticks in OnForexPriceTick ...
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

```
oForex := TsgcWSAPI_Forex.Create(nil);
oForex.Credentials.UserName := '<username>';
oForex.Credentials.Password := '<password>';
oForex.Credentials.AppKey  := '<appkey>';

// explicit REST-only usage
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

```
oForex := TsgcWSAPI_Forex.Create(nil);
oForex.Credentials.UserName := '<username>';
oForex.Credentials.Password := '<password>';
oForex.Credentials.AppKey  := '<appkey>';
oForex.Connect;

oOrder := TsgcHTTPForexTradeOrder.Create;
try
  oOrder.MarketId        := 401484830;
  oOrder.TradingAccountId:= oForex.TradingAccountId;
  oOrder.Direction       := fdBuy;
  oOrder.Quantity        := 1000;
  oOrder.BidPrice        := 1.09120;
  oOrder.OfferPrice      := 1.09130;
```

```
  // attached take-profit at +50 pips (limit)
  oIfDoneTP := TsgcHTTPForexIfDoneOrder.Create;
  oIfDoneTP.Direction := fdSell;
  oIfDoneTP.Quantity  := 1000;
  oIfDoneTP.TriggerPrice := 1.09630;
  oOrder.IfDone.AddLimit(oIfDoneTP);

  // attached stop-loss at -30 pips (stop)
  oIfDoneSL := TsgcHTTPForexIfDoneOrder.Create;
  oIfDoneSL.Direction := fdSell;
  oIfDoneSL.Quantity  := 1000;
  oIfDoneSL.TriggerPrice := 1.08830;
  oOrder.IfDone.AddStop(oIfDoneSL);

  ShowMessage(oForex.NewTradeOrder(oOrder));
finally
  oOrder.Free;
end;
```

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

```
oForex := TsgcWSAPI_Forex.Create(nil);
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

```
oForex := TsgcWSAPI_Forex.Create(nil);
oForex.Credentials.UserName := '<username>';
oForex.Credentials.Password := '<password>';
oForex.Credentials.AppKey  := '<appkey>';
oForex.Connect;

// search for EUR/USD
ShowMessage(oForex.REST_API.ListCfdMarkets('EUR/USD', 0, 10));

// last 100 1-minute bars for market 401484830
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

```
oForex := TsgcWSAPI_Forex.Create(nil);
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;
// ... application ...
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, altems, 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>*); *altems* 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

```
oForex := TsgcWSAPI_Forex.Create(nil);
oForex.Credentials.UserName := '<username>';
oForex.Credentials.Password := '<password>';
oForex.Credentials.AppKey  := '<appkey>';
oForex.OnForexPriceTick := ForexPriceTick;
oForex.Connect;

// EUR/USD market id
oForex.WatchMarket(401484830);

procedure ForexPriceTick(Sender: TObject; const aTick: TsgcWSForexPriceTick);
begin
  // aTick.MarketId, aTick.Bid, aTick.Offer, aTick.Price, aTick.AuditId ...
end;
```

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.

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.

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.
SubscribeOrder-Book	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.
UnSubscribeOrder-Book	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

```
oClient := TsgcWebSocketClient.Create(nil);
oCexPlus := TsgcWSAPI_CexPlus.Create(nil);
oCexPlus.Client := oClient;
oCexPlus.OnCexPlusConnect := OnCexPlusConnectEvent;
oCexPlus.OnCexPlusMessage := OnCexPlusMessageEvent;
oClient.Active := True;

procedure OnCexPlusConnectEvent(Sender: TObject);
begin
  oCexPlus.GetTicker('BTC-USD');
end;

procedure OnCexPlusMessageEvent(Sender: TObject; Event, Msg: string);
begin
  ShowMessage('Ticker data: ' + Msg);
end;
```

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.
NewLimitOrder	Places a new limit order.
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.
GetTransaction-History	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.
FundsDeposit-FromWallet	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.
FundsWithdrawal-ToWallet	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.

```
oClient := TsgcWebSocketClient.Create(nil);
oCexPlus := TsgcWSAPI_CexPlus.Create(nil);
oCexPlus.Client := oClient;
oCexPlus.CexPlus.ApiKey := 'your-api-key';
oCexPlus.CexPlus.ApiSecret := 'your-api-secret';
oCexPlus.OnCexPlusAuthenticated := OnCexPlusAuthenticatedEvent;
oCexPlus.OnCexPlusMessage := OnCexPlusMessageEvent;
oClient.Active := True;

procedure OnCexPlusAuthenticatedEvent(Sender: TObject);
begin
  oCexPlus.GetOrders();
end;

procedure OnCexPlusMessageEvent(Sender: TObject; Event, Msg: string);
begin
  ShowMessage('Orders: ' + Msg);
end;
```

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.

```
procedure OnDiscordDispatch(Sender: TObject; const aEvent, RawData: string);
begin
  DoLog('#discord dispatch: ' + aEvent + ' ' + RawData);
end;
```

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** = 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>.

Transcription Example

Find below an example of real-time transcription using openAI API

```
WSClient := TsgcWebSocketClient.Create(nil);
oAudio := TsgcAudioRecorderWave.Create(nil);
OpenAI := TsgcWSAPI_OpenAI.Create(nil);

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';
```

```
procedure OnOpenAIAudioTranscriptionCompleted(Sender: TObject; const aItem: TsgcWSOpenAIConversation_Item_Complet
begin
  Log('#transcription_completed: ' + aItem.Transcript);
end;
```

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.

```
var
  WSClient: TsgcWebSocketClient;
  MEXC: TsgcWSAPI_MEXC;
begin
  WSClient := TsgcWebSocketClient.Create(nil);
  MEXC := TsgcWSAPI_MEXC.Create(nil);
  MEXC.Client := WSClient;
  MEXC.MEXCAPI.ApiKey := 'YOUR_KEY';
  MEXC.MEXCAPI.ApiSecret := 'YOUR_SECRET';
  MEXC.MEXCUserDataStreams.UserStream := True;
  if WSClient.Connect then
    MEXC.SubscribeTrade('BTCUSDT');
end;
```

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.

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:

```
procedure TMainForm.MEXCMarketStream(Sender: TObject;
  const aMessage: TsgcMEXCSpotProtoMessage; const aStream: TMemoryStream);
var
  Deal: TsgcMEXCDealItem;
begin
  if aMessage.MessageType = sgcmxspmtTrade then
  begin
    Deal := TsgcMEXCTradeStream(aMessage.Trade).PublicDeals.DealsList[0];
    Memo1.Lines.Add(Format('Trade %s x %s', [Deal.Quantity, Deal.Price]));
  end;
end;
```

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.

```
var
  WSClient: TsgcWebSocketClient;
  Futures: TsgcWSAPI_MEXC_Futures;
begin
  WSClient := TsgcWebSocketClient.Create(nil);
  Futures := TsgcWSAPI_MEXC_Futures.Create(nil);
  Futures.Client := WSClient;
  Futures.MEXCAPI.ApiKey := 'YOUR_KEY';
  Futures.MEXCAPI.ApiSecret := 'YOUR_SECRET';
  if WSClient.Connect then
    Futures.SubscribeDepth('BTC_USDT', True);
end;
```

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.

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:

```
procedure TMainForm.MEXCSubscribed(Sender: TObject; const aChannel, aRaw: string);
begin
  Memo1.Lines.Add('Subscribed to ' + aChannel);
end;
procedure TMainForm.MEXCMessage(Sender: TObject; const aChannel, aRaw: string);
begin
  if SameText(aChannel, 'push.depth') then
    Memo1.Lines.Add(aRaw);
end;
```

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/indexPrice**).
GetFairPrice	Returns the fair (mark) price for a contract from **/api/v1/contract/fairPrice**.
GetFundingRate	Reports the latest funding rate for a symbol using **/api/v1/contract/fundingRate**.
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.

Method	Description
GetAccountAssets	Returns margin balances from **/api/v1/private/account/assets**.
GetPositionList	Lists current positions via **/api/v1/private/position/list**, optionally filtered by symbol.
SetPositionLeverage	Updates leverage for a symbol and, if supplied, the margin mode.
PlaceOrder	Places a futures order on **/api/v1/private/order** providing **symbol**, **side**, **positionSide**, **type** and **quantity** plus optional **price**, **clientOrderId** 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** 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 for a position.

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 WebSocket public server.
- **ApiSecret:** it's the secret value of the api.
- **Passphrase:** it's the custom string defined when creating a new api key.

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oBitget := TsgcWS_API_Bitget.Create(nil);
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;
procedure OnConnect(Connection: TsgcWSConnection);
begin
  DoLog('#Bitget Connected');
end;
```

Events

The Bitget 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.
- **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.

```
procedure OnBitgetAuthentication(Sender: TObject; aSuccess: Boolean; const aError, aRawMessage: string);
begin
  if aSuccess then
  begin
    oBitget.SubscribeOrders('BTCUSDT');
    oBitget.SubscribeAccount();
  end;
end;
```

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.

```
oBitget := TsgcWS_API_Bitget.Create(nil);
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.

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oGateIO := TsgcWS_API_GateIO.Create(nil);
oGateIO.Client := oClient;
oGateIO.GateIO.ApiKey := 'your_api_key';
oGateIO.GateIO.ApiSecret := 'your_api_secret';
oGateIO.GateIOChannel := giocSpot;
oClient.Active := True;
procedure OnConnect(Connection: TsgcWSConnection);
begin
  DoLog('#GateIO Connected');
end;
```

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.

```
procedure OnConnect(Connection: TsgcWSConnection);
begin
  oGateIO.SubscribeTicker('BTC_USDT');
  oGateIO.SubscribeOrderBook('BTC_USDT', '20', '100ms');
end;
```

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.

```
oGateIO := TsgcWS_API_GateIO.Create(nil);
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.

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oDeribit := TsgcWS_API_Deribit.Create(nil);
oDeribit.Client := oClient;
oDeribit.Deribit.ApiKey := 'your_api_key';
oDeribit.Deribit.ApiSecret := 'your_api_secret';
oDeribit.Deribit.TestNet := False;
oClient.Active := True;
procedure OnConnect(Connection: TsgcWSConnection);
begin
  DoLog('#Deribit Connected');
end;
```

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.

```
procedure OnDeribitAuthentication(Sender: TObject; aSuccess: Boolean; const aError, aRawMessage: string);
begin
  if aSuccess then
  begin
    oDeribit.SubscribeOrders('BTC-PERPETUAL');
    oDeribit.SubscribePositions('BTC');
  end;
end;
```

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.

```
oDeribit := TsgcWS_API_Deribit.Create(nil);
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.

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oCryptoCom := TsgcWS_API_CryptoCom.Create(nil);
oCryptoCom.Client := oClient;
oCryptoCom.CryptoCom.ApiKey := 'your_api_key';
oCryptoCom.CryptoCom.ApiSecret := 'your_api_secret';
oCryptoCom.Channel := cccMarket;
oClient.Active := True;
procedure OnConnect(Connection: TsgcWSConnection);
begin
  DoLog('#CryptoCom Connected');
end;
```

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.

```
procedure OnConnect(Connection: TsgcWSConnection);
begin
  oCryptoCom.SubscribeTicker('BTC_USDT');
  oCryptoCom.SubscribeBook('BTC_USDT', '10');
end;
```

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.

```
oCryptoCom := TsgcWS_API_CryptoCom.Create(nil);
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.

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oHTX := TsgcWS_API_Huobi.Create(nil);
oHTX.Client := oClient;
oHTX.Huobi.ApiKey := 'your_api_key';
oHTX.Huobi.ApiSecret := 'your_api_secret';
oClient.Active := True;
procedure OnConnect(Connection: TsgcWSConnection);
begin
  DoLog('#HTX Connected');
end;
```

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.

```
oHTX := TsgcHTTP_API_HTX.Create(nil);
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**.

```
oClient := TsgcWhatsapp_Client.Create(nil);
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.

```
oClient := TsgcWhatsapp_Client.Create(nil);
oClient.ServerOptions.WebhookOptions.Endpoint := '/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.

```
procedure OnMessageReceived(Sender: TObject; const aMessage: TsgcWhatsapp_Receive_Message; var aMarkAsRead: Boole
begin
  DoLog('Received: ' + aMessage.Messages._Message[0].Id);
end;
```

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.

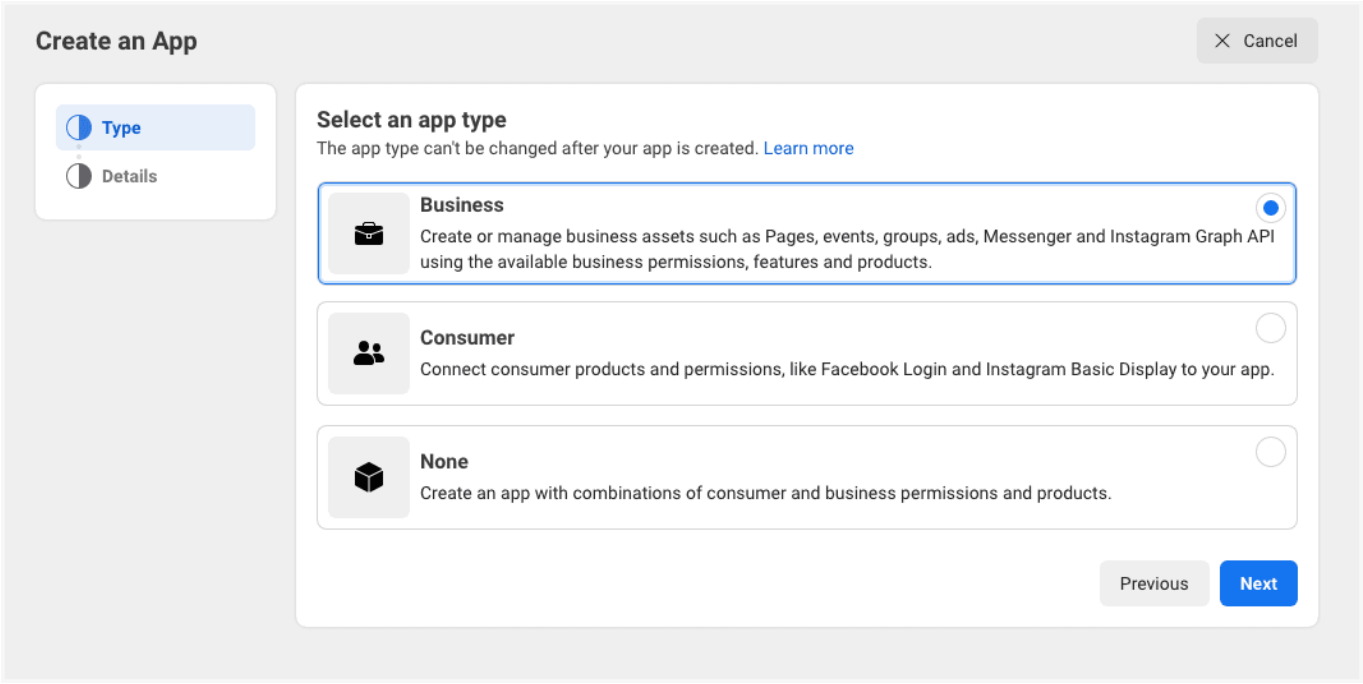
OnMessageSent

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.

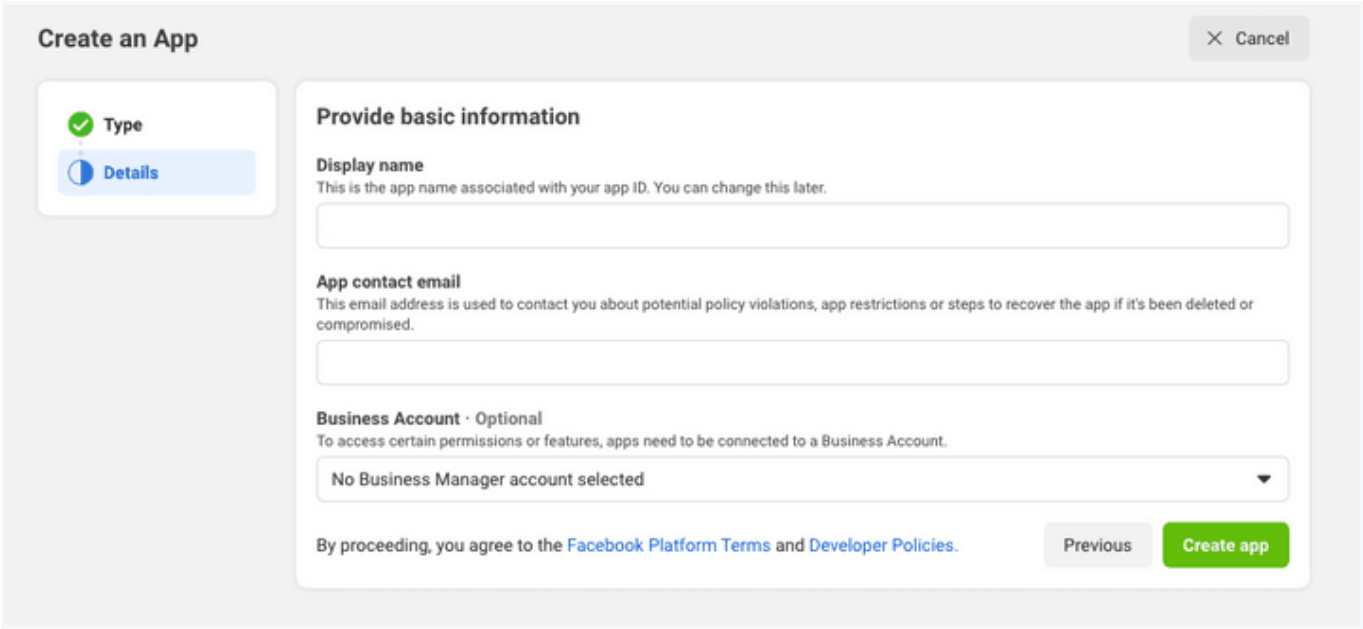
WhatsApp Create App

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.

```
oClient := TsgcWhatsapp_Client.Create(nil);
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.

```
oClient := TsgcWhatsapp_Client.Create(nil);
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.

```
oClient := TsgcWhatsapp_Client.Create(nil);
oClient.ServerOptions.WebhookOptions.Endpoint := '/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.

```
oClient := TsgcWhatsApp_Client.Create(nil);
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).

```
oClient := TsgcWhatsApp_Client.Create(nil);
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).

```
oClient := TsgcWhatsApp_Client.Create(nil);
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

```
oClient := TsgcWhatsApp_Client.Create(nil);
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

```
oClient := TsgcWhatsApp_Client.Create(nil);
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

```
oClient := TsgcWhatsApp_Client.Create(nil);
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.

```
oClient := TsgcWhatsApp_Client.Create(nil);
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).

```
oClient := TsgcWhatsApp_Client.Create(nil);
oClient.WhatsappOptions.PhoneNumberId := '107809351952205';
oClient.WhatsappOptions.Token := 'EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2';
oClient.SendMessageContact('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

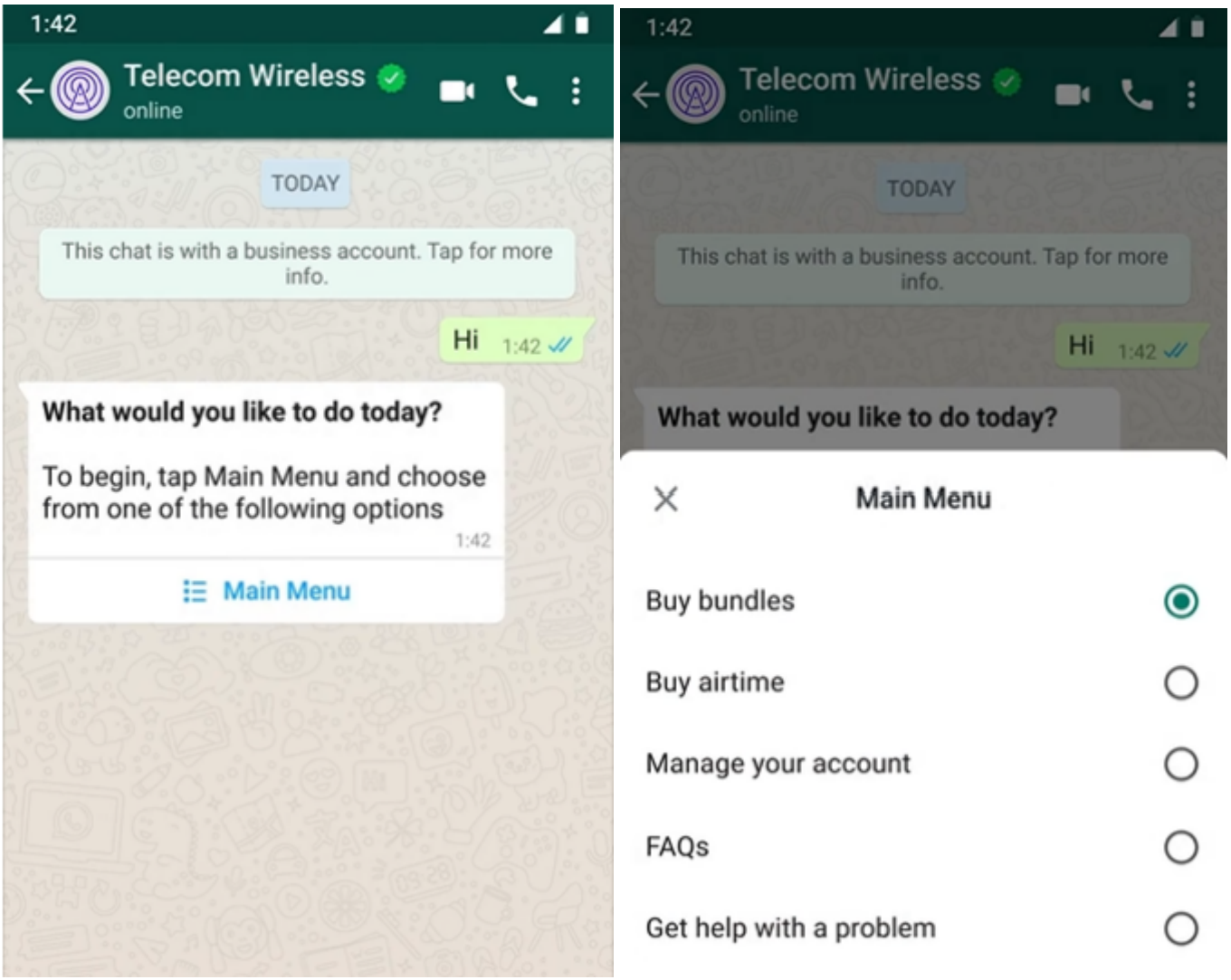
Reply Buttons are best for offering quick responses from a limited set of options, such as:

- 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.

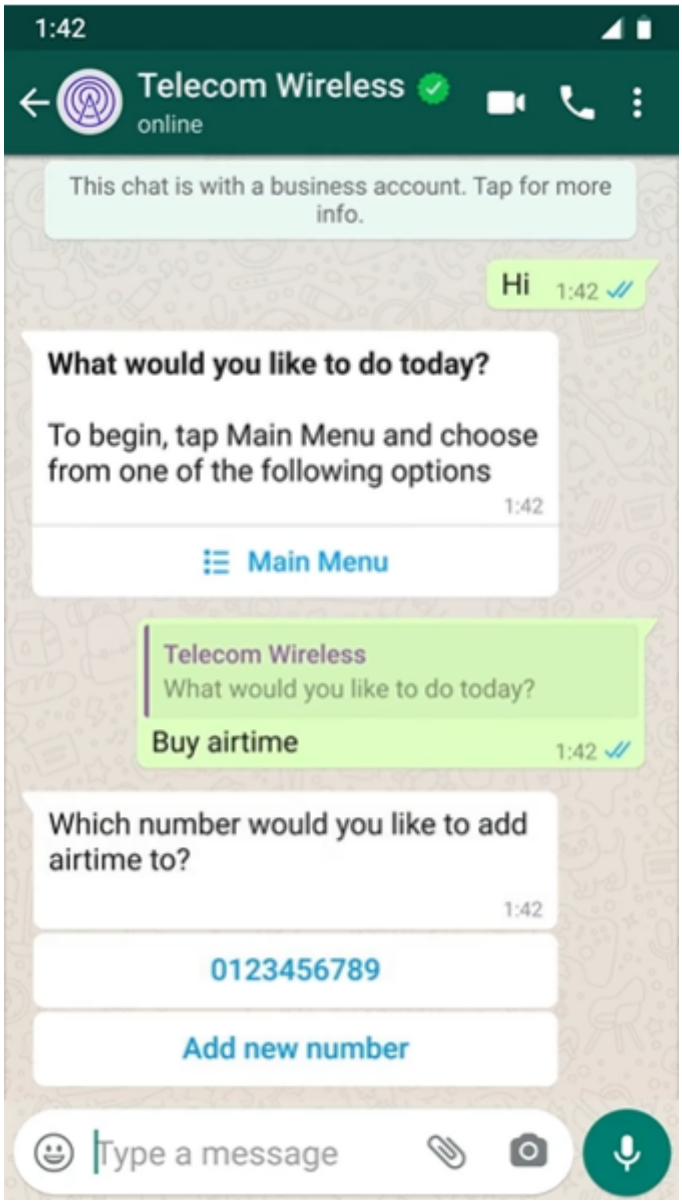
Interactive List

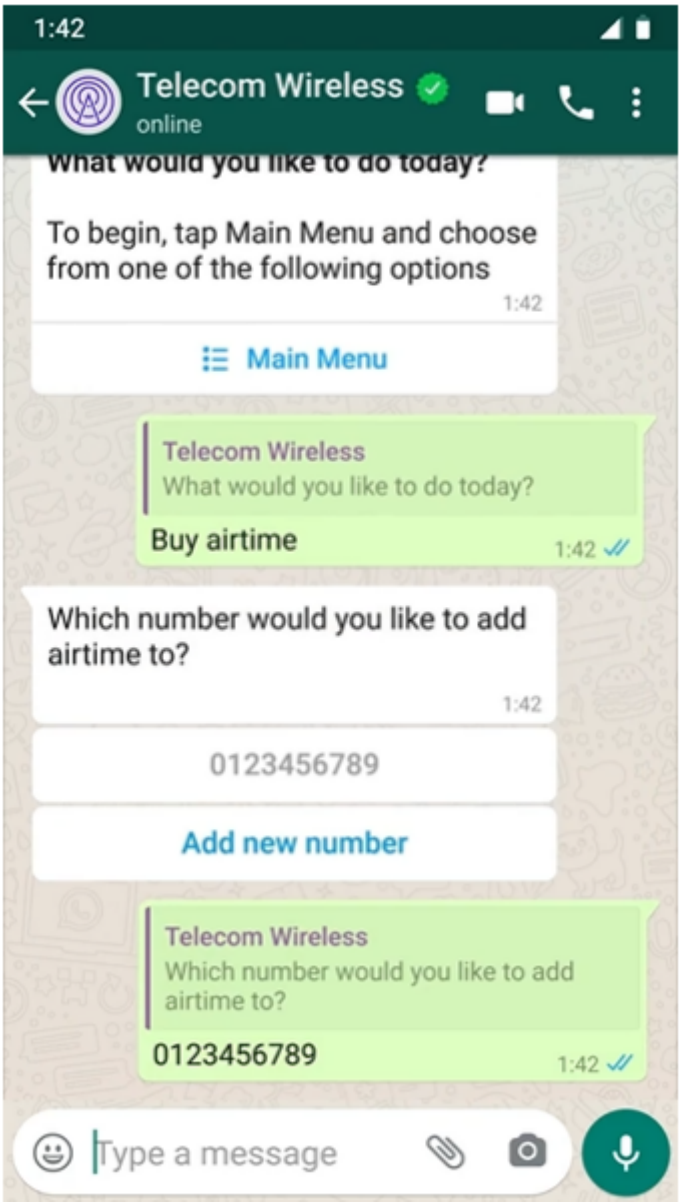
```
oClient := TsgcWhatsApp_Client.Create(nil);
oClient.WhatsappOptions.PhoneNumberId := '107809351952205';
oClient.WhatsappOptions.Token := 'EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2';
oClient.SendMessageInteractiveList('34605889421', 'What Would you like to do today?', 'To begin, Tap Main Menu ar
```



Reply Buttons

```
oClient := TsgcWhatsApp_Client.Create(nil);
oClient.WhatsappOptions.PhoneNumberId := '107809351952205';
oClient.WhatsappOptions.Token := 'EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2';
oClient.SendMessageInteractiveButtons('34605889421', 'Select an option', 'Which number would you like to add airt
```


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.

```
oClient := TsgcWhatsApp_Client.Create(nil);
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

```
procedure SendSamplePurchaseFeedbackTemplate(const aName: string);
var
  oTemplate: TsgcWhatsApp_Send_Message_Template;
  oComponent: TsgcWhatsApp_Send_Message_Template_Component;
  oParameter: TsgcWhatsApp_Send_Message_Template_Parameter;
begin
  oTemplate := TsgcWhatsApp_Send_Message_Template.Create;
  Try
    oTemplate.Language.Code := 'en_US';
    oTemplate.TemplateName := 'sample_purchase_feedback';
    // ... header
    oComponent := TsgcWhatsApp_Send_Message_Template_Component.Create;
    oComponent._Type := wapctHeader;
    oTemplate.Components.Add(oComponent);
    oParameter := TsgcWhatsApp_Send_Message_Template_Parameter.Create;
    oParameter.Image.Link := 'https://www.esegece.com/images/esegece.png';
    oParameter._Type := wapptImage;
    oComponent.Parameters.Add(oParameter);
    // ... body
    oComponent := TsgcWhatsApp_Send_Message_Template_Component.Create;
    oComponent._Type := wapctBody;
    oTemplate.Components.Add(oComponent);
    oParameter := TsgcWhatsApp_Send_Message_Template_Parameter.Create;
    oParameter.Text := aName;
    oParameter._Type := wapptText;
    oComponent.Parameters.Add(oParameter);
    whatsapp.SendMessageTemplate('107809351952205', oTemplate);
  Finally
    oTemplate.Free;
  End;
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.

```
procedure SendSamplePurchaseFeedbackTemplate(const aName: string);
var
  oTemplate: TsgcWhatsApp_Send_Message_Template;
  oComponent: TsgcWhatsApp_Send_Message_Template_Component;
  oParameter: TsgcWhatsApp_Send_Message_Template_Parameter;
  vId: string;
begin
  // ... first upload the file
  vId := whatsapp.UploadMedia('c:\images\file.png', 'image/png');
```

```
  // ... send message
  oTemplate := TsgcWhatsApp_Send_Message_Template.Create;
  Try
    oTemplate.Language.Code := 'en_US';
    oTemplate.TemplateName := 'sample_purchase_feedback';
    // ... header
    oComponent := TsgcWhatsApp_Send_Message_Template_Component.Create;
    oComponent._Type := wapctHeader;
    oTemplate.Components.Add(oComponent);
    oParameter := TsgcWhatsApp_Send_Message_Template_Parameter.Create;
    oParameter.Image.id := vId;
    oParameter._Type := wapptImage;
    oComponent.Parameters.Add(oParameter);
    // ... body
    oComponent := TsgcWhatsApp_Send_Message_Template_Component.Create;
    oComponent._Type := wapctBody;
    oTemplate.Components.Add(oComponent);
    oParameter := TsgcWhatsApp_Send_Message_Template_Parameter.Create;
    oParameter.Text := aName;
    oParameter._Type := wapptText;
    oComponent.Parameters.Add(oParameter);
    whatsapp.SendMessageTemplate('107809351952205', oTemplate);
  Finally
    oTemplate.Free;
  End;
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.

```
procedure OnWhatsAppMessageReceived(Sender: TObject; const aMessage: TsgcWhatsApp_Receive_Message; var aMarkAsRea
var
  vText: string;
  vTo: string;
begin
  if aMessage.Contacts.Count > 0 then
  begin
    vTo := aMessage.Contacts.Contact[0].WaID;
    if aMessage.Messages.Count > 0 then
    begin
      if aMessage.Messages._Message[0]._Type = wapmrtText then
      begin
        vText := 'ECHO ==> ' + aMessage.Messages._Message[0].Text.Body;
        WhatsApp.SendMessageText(vTo, vText);
        aMarkAsRead := True;
      end;
    end;
  end;
end;
```

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.

```
procedure OnWhatsAppMessageSent(Sender: TObject; const aMessage: TsgcWhatsApp_Receive_Message; aStatus: TsgcWhats
begin
  vPhone := aMessage.MetaData.DisplayPhoneNumber;
  case aStatus of
    wapsmstSent: DoLog('Message to ' + vPhone + ' sent.');
    wapsmstDelivered: DoLog('Message to ' + vPhone + ' delivered.');
    wapsmstRead: DoLog('Message to ' + vPhone + ' read.');
    else
      DoLog('Message to ' + vPhone + ' unknown status.')
  end;
end;
```

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).

```
oClient := TsgcWhatsApp_Client.Create(nil);
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).

```
oClient := TsgcWhatsApp_Client.Create(nil);
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

```
oClient := TsgcWhatsApp_Client.Create(nil);
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

```
oClient := TsgcWhatsApp_Client.Create(nil);
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

```
oClient := TsgcWhatsApp_Client.Create(nil);
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**.

```
oClient := TsgcWhatsApp_Client.Create(nil);
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

```
oTelegram := TsgcTDLib_Telegram.Create(nil);
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.

```
procedure OnAuthenticationCode(Sender: TObject; var Code: string);
begin
  Code := InputBox('Telegram Code', 'Introduce code', '');
end;
```

In Android, inputbox doesn't locks the thread, so instead of return the value introduced by user in Code parameter, use the method SetAuthenticationCode to set the code value.

```
procedure OnAuthenticationCode(Sender: TObject; var Code:string);
begin
  InputBox('Telegram', 'Introduce Telegram Code', '',
    procedure(const AResult: TModalResult; const AValue: string)
    begin
      sgcTelegram.SetAuthenticationCode(AValue);
    end
    );
end;
```

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.
GetChat-Member	**aChatId:** Chat Identifier. **aUserId:** User Identifier.	Returns information about a single member of a chat.
GetBasic-Group-FullInfo	**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.

AddProxyHTTP	**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.
AddProxyMTProto	**aServer:** Server name of proxy. **aPort:** Number of proxy port. **aSecret:** The proxy's secret in hexadecimal encoding.	Adds a MTProto proxy server for network requests. Can be called before authorization.
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

```
oTelegram := TsgcTDLib_Telegram.Create(nil);
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.

```
oTelegram := TsgcTDLib_Telegram.Create(nil);
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.

```
oTelegram := TsgcTDLib_Telegram.Create(nil);
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;
procedure OnAuthenticationCode(Sender: TObject; var Code: string);
begin
  Code := InputBox('Telegram Code', 'Introduce code', '');
end;
procedure OnMessageText(Sender: TObject; MessageText: TsgcTelegramMessageText);
begin
```

```
  Log('Message Received: ' + MessageText.Text);
end;
procedure OnConnectionStatus(Sender: TObject; const Status: string);
begin
  if Status = 'connectionStateReady' then
    oTelegram.SendTextMessage('1234', 'Hello Telegram!');
end;
```

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.

```
oReplyMarkup := TsgcTelegramReplyMarkupInlineKeyboard.Create;
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;
End;

      procedure OnNewCallbackQuery(Sender: TObject; CallbackQuery: TsgcTelegramCallbackQuery);
begin
  if CallbackQuery.PayloadData.Data = 'I like it' then
    ShowMessage('yes')
  else
    ShowMessage('no');
end;
```

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 := TsgcTelegramReplyMarkupShowKeyboard.Create;
Try
  oReplyMarkup.AddButtonTypeRequestPhoneNumber('Give me your phone');
  sgcTelegram.SendTextMessage('123456', 'Please provide the information below', nil, oReplyMarkup);
Finally
  oReplyMarkup.Free;
End;
```

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**.

```
oTelegram := TsgcTDLib_Telegram.Create(nil);
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.

```
procedure(Sender: TObject; MessageSponsored: TsgcTelegramMessageSponsored);
begin
  DoLog(MessageSponsored.Text);
end;
```

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**.

```
procedure SendInvoice;
var
  oInvoice: TsgcTelegramSendInvoice;
begin
  oInvoice := TsgcTelegramSendInvoice.Create;
  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;
  End;
end;
```

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);


procedure OnTelegramEvent(Sender: TObject; const Event, Text: string);
begin
  if Event = 'chatMembers' then
    ReadJSON(Text);
end;
```

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.

```
oTelegram := TsgcTDLib_Telegram.Create(nil);
oTelegram.Telegram.API.ApiHash := 'ABCDEFGHIJKLMN';
oTelegram.Telegram.API.ApiId := '1234';
oTelegram.PhoneNumber := '008745744155';
oTelegram.Active := true;

procedure OnTelegramAuthenticationCode(Sender: TObject; var Code: string);
begin
  Code := 'code sent to phone';
end;

procedure OnTelegramRegisterUser(Sender: TObject; var FirstName, LastName: string);
begin
  FirstName := 'first name';
  LastName := 'last name';
end;
```

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.

```
oRCON := TsgcLib_RCON.Create(nil);
oRCON.RCON_Options.Host := '127.0.0.1';
oRCON.RCON_Options.Port := 25575;
oRCON.RCON_Options.Password := 'test';
oRCON.Active := True;


procedure OnAuthenticate(Sender: TObject; Authenticated: Boolean; const aPacket: TsgcRCON_Packet);
begin
  if Authenticated then
    DoLog('#authenticated')
  else
    DoLog('#not authenticated');
end;


procedure OnResponse(Sender: TObject; const aResponse: string; const aPacket: TsgcRCON_Packet);
begin
  DoLog(aResponse);
end;
```

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.

```
oCryptoHopper := TsgcHTTP_Cryptohopper.Create(nil);
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.

```
function EnableHopper: string;
var
  oHopper: TsgcHTTPCTHopper;
begin
  oHopper := TsgcHTTPCTHopper.Create;
  Try
    if Cryptohopper.GetHopper('1234', oHopper) then
    begin
      oHopper.Enabled := 1;
      result := Cryptohopper.UpdateHopper('1234', oHopper);
    end;
  Finally
    FreeAndNil(oHopper);
  End;
end;
```

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;
```

RTCMultiConnection

RTCMultiConnection

RTCMultiConnection is a WebRTC JavaScript library for peer-to-peer applications (screen sharing, audio/video conferencing, file sharing, media streaming etc.)

Configuration

The RTCMultiConnection requires a WebSocket server for Signaling, so link the server property of RTCMultiConnection to a WebSocket Server (like TsgcWebSocketHTTPServer). Find below the properties you must configure.

Server

Host: is the public IP address or DNS name of WebSocket server.
Port: is the listening port of WebSocket Server.

IceServers

Is the configuration of the ICE servers (STUN/TURN) to allow communication between peers. Example:

```
[
  {
     "urls": "stun:www.yourstun.com"},
  {
    "urls": "turn:www.yourturn.com",
    "username": "user",
    "credential": "secret"
  }
]
```

VideoResolution

Here you can configure the Video Resolution of Video Conferences, the higher the resolution, the more bandwidth is required by the connection.

HTMLDocuments

Configure for every Application which is the name of the HTML page that serves this content.

Example: if the server is running on website www.webrtc.com on port 8443 and the HTMLDocuments.VideoConferencing = /RTCMultiConnection-VideoConferencing.html, the url to access the VideoConferencing will be

```
https://www.webrtc.com:8443/RTCMultiConnection-VideoConferencing.html
```

WebRTC requires a secure connection (HTTPs) so requires the use of certificates, read more Server SSL.

Applications

Name	Description
VideoConferencing	Multi-user (many-to-many) video chat using mesh networking model.
ScreenSharing	Multi-user (one-to-many) screen sharing using star topology.

Video-Broad-casting	Multi-user (one-to-many) video broadcasting using star topology.

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.

```
procedure SendWebPushNotification;
var
  oSubscription: TsgcHTTP_API_WebPush_PushSubscription;
  oWebPush: TsgcWebPush_Client ;
begin
  oSubscription := TsgcHTTP_API_WebPush_PushSubscription.Create;
  try
    oSubscription.Endpoint := 'endpoint';
    oSubscription.PublicKey := 'public key';
    oSubscription.AuthSecret := 'authentication secret';
    oWebPush := TsgcHTTP_API_WebPush_Client.Create(nil);
    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": "Hello from eSeGeCe
    finally
      oWebPush.Free;
    end;
  finally
    oSubscription.Free;
  end;
end;
```

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.

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 the server, **MCPOptions.SessionTimeout** to control session lifetime 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`.

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.

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.

```
procedure OnMCPSessionEnd(Sender: TObject; const aSession:
    TsgcAI_MCP_Session);
begin
  WriteLn('#session_end: ' + aSession.Id);
end;
procedure OnMCPSessionNew(Sender: TObject; const aSession:
    TsgcAI_MCP_Session);
begin
  WriteLn('#session_new: ' + aSession.Id);
end;
```

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:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  oTool: TsgcAI_MCP_Tool;
begin
  oTool := MCPServer.Tools.AddTool('math.add', 'Adds two numbers');
  oTool.InputSchema.Properties.AddProperty('a', True, aimcpjtNumber, 'Left operand');
  oTool.InputSchema.Properties.AddProperty('b', True, aimcpjtNumber, 'Right operand');
end;
```

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" }
          }
        }
      }
    ]
  }
}
```

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:

```
procedure TMainForm.MCPServerMCPRequestTool(Sender: TObject;
  const ASession: TsgcAI_MCP_Session;
  const ARequest: TsgcAI_MCP_Request_ToolsCall;
  const AResponse: TsgcAI_MCP_Response_ToolsCall);
var
  LA, LB: Double;
begin
  if ARequest.Params.Name = 'math.add' then
  begin
    LA := ARequest.Params.Arguments.Node['a'].AsNumber;
    LB := ARequest.Params.Arguments.Node['b'].AsNumber;
    AResponse.Result.Content.AddText(Format('Sum = %.2f', [LA + LB]));
  end
  else
    AResponse.Result.IsError := True;
end;
```

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.

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.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.

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.

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  LTool: TsgcAI_MCP_Tool;
begin
  MCPServer.Tools.Clear;
  LTool := MCPServer.Tools.AddTool('geo.locate', 'Returns coordinates for a city and country');
  LTool.InputSchema.Properties.AddProperty('city', True, aimcpjtString, 'City to search');
  LTool.InputSchema.Properties.AddProperty('country', False, aimcpjtString, 'Optional ISO country code');
end;
procedure TMainForm.MCPServerMCPRequestTool(Sender: TObject;
  const ASession: TsgcAI_MCP_Session;
  const ARequest: TsgcAI_MCP_Request_ToolsCall;
  const AResponse: TsgcAI_MCP_Response_ToolsCall);
var
  LCity, LCountry: string;
begin
  if not SameText(ARequest.Params.Name, 'geo.locate') then
    Exit;
  LCity := ARequest.Params.Arguments.Node['city'].AsString;
  LCountry := ARequest.Params.Arguments.Node['country'].AsString;
  if LCity = '' then
  begin
    AResponse.Result.Content.Clear;
    AResponse.Result.Content.AddText('City cannot be empty.');
    AResponse.Result.IsError := True;
    Exit;
  end;
  AResponse.Result.Content.Clear;
  AResponse.Result.Content.AddText('City: ' + LCity);
  if LCountry <> '' then
    AResponse.Result.Content.AddText('Country: ' + LCountry);
  AResponse.Result.Content.AddText('Coordinates: ' + LookupCityCoordinates(LCity, LCountry));
end;
```

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:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  oPrompt: TsgcAI_MCP_Prompt;
begin
  MCPServer.Prompts.Clear;
  oPrompt := MCPServer.Prompts.AddPrompt('CodeReview',
    'Asks the LLM to analyze code quality and suggest improvements');
  oPrompt.Arguments.AddArgument('code', 'The code to review', True);
end;
```

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:

```
procedure TFRMMCPServer.MCPServerMCPRequestPrompt(Sender: TObject;
  const aSession: TsgcAI_MCP_Session; const aRequest: TsgcAI_MCP_Request_PromptsGet;
  const aResponse: TsgcAI_MCP_Response_PromptsGet);
begin
  if aRequest.Params.Name = 'CodeReview' then
  begin
    aResponse.Result.Description := 'Code review prompt';
    aResponse.Result.Messages.AddText('user',
      'Please review this Delphi code: ShowMessage(''Hello World'' ');
  end;
```

```
end;
```

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.

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  LPrompt: TsgcAI_MCP_Prompt;
begin
  MCPServer.Prompts.Clear;
  LPrompt := MCPServer.Prompts.AddPrompt('IssueTriage', 'Collect details before escalating support tickets');
  LPrompt.Arguments.Clear;
  LPrompt.Arguments.AddArgument('summary', 'One-line description supplied by the user', True);
  LPrompt.Arguments.AddArgument('customerPriority', 'Optional priority label', False);
  LPrompt.Messages.AddText('system', 'You are a support assistant that triages technical issues.');
end;
procedure TMainForm.MCPServerMCPRequestPrompt(Sender: TObject;
  const aSession: TsgcAI_MCP_Session; const aRequest: TsgcAI_MCP_Request_PromptsGet;
  const aResponse: TsgcAI_MCP_Response_PromptsGet);
begin
```

```
  if not SameText(aRequest.Params.Name, 'IssueTriage') then
    Exit;
  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 <> '' then
    aResponse.Result.Messages.AddText('assistant', 'Adjust SLA checks based on the provided customer priority.');
end;
```

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:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  MCPServer.Resources.Clear;
  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');
end;
```

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:

```
procedure OnMCPRequestResource(Sender: TObject;
  const aSession: TsgcAI_MCP_Session; const aRequest: TsgcAI_MCP_Request_ResourcesRead;
  const aResponse: TsgcAI_MCP_Response_ResourcesRead);
begin
  if aRequest.Params.Uri = 'file:///project/src/main.rs' then
    aResponse.Result.Contents.AddContentText('file:///project/src/main.rs',
      'main.rs', 'Rust Software Application Main File', 'text/x-rust',
```

```
      'fn main() {\n    println!(\"Hello world!\");\n}');
end;
```

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.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  MCPServer.Resources.Clear;
  MCPServer.Resources.AddResource('metrics://daily', 'DailyMetrics', 'Daily KPI snapshot', 'Summaries generated e
end;
procedure TMainForm.MCPServerMCPRequestResource(Sender: TObject;
  const aSession: TsgcAI_MCP_Session; const aRequest: TsgcAI_MCP_Request_ResourcesRead;
  const aResponse: TsgcAI_MCP_Response_ResourcesRead);
var
  LTargetUri: string;
begin
  LTargetUri := aRequest.Params.Uri.ToLower;
  aResponse.Result.Contents.Clear;
  if LTargetUri = 'metrics://daily' then
  begin
    aResponse.Result.Contents.AddContentText('metrics://daily', 'DailyMetrics', 'Daily KPI snapshot', 'applicatio
    Exit;
  end;
  aResponse.Result.IsError := True;
  aResponse.Result.Contents.AddContentText(aRequest.Params.Uri, 'UnknownResource', 'Not found', 'text/plain', 'Re
end;
```

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.

Sample code

The snippets below request roots as soon as a session is established and log the returned entries.

```
procedure TFRMMCPServer.FormCreate(Sender: TObject);
begin
  MCPServer.OnMCPSessionNew := MCPServerSessionNew;
  MCPServer.OnMCPResponseRootsList := MCPServerRootsList;
end;

procedure TFRMMCPServer.MCPServerSessionNew(Sender: TObject;
  const aSession: TsgcAI_MCP_Session);
begin
  if aSession.ClientCapabilities.Roots.Enabled then
    MemoLog.Lines.Add('roots/list sent id ' +
      MCPServer.RequestRootsList(aSession));
end;

procedure TFRMMCPServer.MCPServerRootsList(Sender: TObject;
  const aSession: TsgcAI_MCP_Session;
  const aRequest: TsgcAI_MCP_Request_RootsList;
  const aResponse: TsgcAI_MCP_Response_RootsList);
begin
  MemoLog.Lines.Add(Format('roots/list returned %d entries',
    [aResponse.Roots.Count]));
  MemoLog.Lines.Add('roots/list payload ' + aResponse.Write);
end;
```

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.

Sample code

The snippets below request a short summary from the client and print the assistant reply.

```
procedure TFRMMCPServer.FormCreate(Sender: TObject);
begin
  MCPServer.OnMCPResponseSamplingCreateMessage := MCPServerSamplingResponse;
end;
procedure TFRMMCPServer.RequestTicketSummary(
  const aSession: TsgcAI_MCP_Session; const aTicketText: string);
var
  LRequest: TsgcAI_MCP_Request_SamplingCreateMessage;
begin
  if not aSession.ClientCapabilities.Sampling.Enabled then
    Exit;
  LRequest := TsgcAI_MCP_Request_SamplingCreateMessage.Create;
  try
    LRequest.Params.Messages.AddTextMessage('user',
      'Summarise the following ticket: ' + aTicketText);
    LRequest.Params.SystemPrompt := 'You are a concise support assistant.';
    LRequest.Params.ModelPreferences.Hints.AddHint('claude-3-sonnet');
    LRequest.Params.ModelPreferences.SpeedPriority := 0.6;
    LRequest.Params.ModelPreferences.IntelligencePriority := 0.8;
    LRequest.Params.HasMaxTokens := True;
    LRequest.Params.MaxTokens := 200;
    MemoLog.Lines.Add('sampling/createMessage id ' +
      MCPServer.RequestSamplingCreateMessage(aSession, LRequest));
  finally
    LRequest.Free;
  end;
end;
procedure TFRMMCPServer.MCPServerSamplingResponse(Sender: TObject;
  const aSession: TsgcAI_MCP_Session;
  const aRequest: TsgcAI_MCP_Request_SamplingCreateMessage;
  const aResponse: TsgcAI_MCP_Response_SamplingCreateMessage);
var
  LText: TsgcAI_MCP_Response_Result_Content_Text;
begin
  if aResponse.Content is TsgcAI_MCP_Response_Result_Content_Text then
  begin
    LText := TsgcAI_MCP_Response_Result_Content_Text(aResponse.Content);
    MemoLog.Lines.Add('assistant: ' + LText.Text);
  end;
  MemoLog.Lines.Add('model: ' + aResponse.Model +
    ' reason: ' + aResponse.StopReason);
end;
```

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.

Sample code

The snippets below request contact information, wait for the reply, and log the resulting action plus payload.

```
procedure TFRMMCPServer.FormCreate(Sender: TObject);
begin
  MCPServer.OnMCPResponseElicitationCreate := MCPServerElicitationResponse;
end;
procedure TFRMMCPServer.RequestContactDetails(
  const aSession: TsgcAI_MCP_Session);
var
  LRequest: TsgcAI_MCP_Request_ElicitationCreate;
  LField: TsgcAI_MCP_Elicitation_Schema_Property;
begin
  if not aSession.ClientCapabilities.Elicitation.Enabled then
    Exit;
  LRequest := TsgcAI_MCP_Request_ElicitationCreate.Create;
  try
    LRequest.Params._Message := 'Please confirm your contact information.';
    LRequest.Params.RequestedSchema.Title := 'Contact information';
    LRequest.Params.RequestedSchema.Description :=
      'Used to keep you updated about this ticket.';
    LField := LRequest.Params.RequestedSchema.Properties.AddProperty('name');
    LField._Type := 'string';
    LField.Title := 'Full name';
    LField.Required := True;
    LField := LRequest.Params.RequestedSchema.Properties.AddProperty('email');
    LField._Type := 'string';
    LField.Title := 'Email address';
    LField.Format := 'email';
    LField.Required := True;
    LField := LRequest.Params.RequestedSchema.Properties.AddProperty('updates');
    LField._Type := 'boolean';
    LField.Title := 'Receive status updates';
    LField.Description := 'Tick to receive notifications when the ticket changes.';
    LField.DefaultBoolean := False;
    MemoLog.Lines.Add('elicitation/create id ' +
      MCPServer.RequestElicitationCreate(aSession, LRequest));
  finally
    LRequest.Free;
  end;
end;
procedure TFRMMCPServer.MCPServerElicitationResponse(Sender: TObject;
  const aSession: TsgcAI_MCP_Session;
  const aRequest: TsgcAI_MCP_Request_ElicitationCreate;
  const aResponse: TsgcAI_MCP_Response_ElicitationCreate);
begin
  MemoLog.Lines.Add('elicitation action: ' + aResponse.Action);
```

```
  if SameText(aResponse.Action, 'accept') and Assigned(aResponse.Content) then
    MemoLog.Lines.Add('payload: ' + aResponse.Content.Text);
end;
```

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.

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.
- **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 — 2 properties available on the component.
- Methods — 12 public methods.
- Events — 12 events you can handle.

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.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MCP.OnMCPListTools := MCPListTools;
  MCP.OnMCPResponseTool := MCPResponseTool;
end;
procedure TForm1.LoadTools;
begin
  MCP.ListTools;
end;
procedure TForm1.MCPListTools(Sender: TObject;
  const ARequest: TsgcAI_MCP_Request_ToolsList;
  const AResponse: TsgcAI_MCP_Response_ToolsList);
var
  LIndex: Integer;
  LTool: TsgcAI_MCP_Tool;
begin
  for LIndex := 0 to AResponse.Tools.Count - 1 do
  begin
    LTool := TsgcAI_MCP_Tool(AResponse.Tools.Item[LIndex]);
    Memo1.Lines.Add(Format('%s (%s)', [LTool.Name, LTool.Description]));
  end;
end;
procedure TForm1.CallTemperatureTool;
var
  LArgs: TsgcJSON;
begin
  LArgs := TsgcJSON.Create(nil);
  try
    LArgs.AddPair('city', 'Barcelona');
    MCP.RequestTool('GetTemperature', LArgs);
  finally
    LArgs.Free;
  end;
```

```
end;
procedure TForm1.MCPResponseTool(Sender: TObject;
  const ARequest: TsgcAI_MCP_Request_ToolsCall;
  const AResponse: TsgcAI_MCP_Response_ToolsCall);
var
  LIndex: Integer;
  LContent: TsgcAI_MCP_Response_Result_Content;
begin
  if AResponse.Result.IsError then
    Memo1.Lines.Add('Tool returned an error payload');
  for LIndex := 0 to AResponse.Result.Content.Count - 1 do
  begin
    LContent := TsgcAI_MCP_Response_Result_Content(AResponse.Result.Content.Item[LIndex]);
    Memo1.Lines.Add(LContent.Write);
  end;
end;
```

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.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MCP.OnMCPListPrompts := MCPListPrompts;
  MCP.OnMCPResponsePrompt := MCPResponsePrompt;
end;
procedure TForm1.LoadPrompts;
begin
  MCP.ListPrompts;
end;
procedure TForm1.MCPListPrompts(Sender: TObject;
  const ARequest: TsgcAI_MCP_Request_PromptsList;
  const AResponse: TsgcAI_MCP_Response_PromptsList);
var
  LIndex: Integer;
  LPrompt: TsgcAI_MCP_Prompt;
begin
  for LIndex := 0 to AResponse.Prompts.Count - 1 do
  begin
    LPrompt := TsgcAI_MCP_Prompt(AResponse.Prompts.Item[LIndex]);
    Memo1.Lines.Add(Format('%s: %s', [LPrompt.Name, LPrompt.Description]));
  end;
end;
procedure TForm1.ExecutePrompt;
var
  LArgs: TsgcJSON;
begin
  LArgs := TsgcJSON.Create(nil);
  try
    LArgs.AddPair('code', 'ShowMessage(''Hello World'')');
    MCP.RequestPrompt('CodeReview', LArgs);
  finally
    LArgs.Free;
```

```
  end;
end;
procedure TForm1.MCPResponsePrompt(Sender: TObject;
  const ARequest: TsgcAI_MCP_Request_PromptsGet;
  const AResponse: TsgcAI_MCP_Response_PromptsGet);
var
  LIndex: Integer;
  LMessage: TsgcAI_MCP_Response_PromptsGet_Result_Message;
begin
  Memo1.Lines.Add('Prompt description: ' + AResponse.Result.Description);
  for LIndex := 0 to AResponse.Result.Messages.Count - 1 do
  begin
    LMessage := TsgcAI_MCP_Response_PromptsGet_Result_Message(AResponse.Result.Messages.Item[LIndex]);
    if LMessage is TsgcAI_MCP_Response_PromptsGet_Result_Message_Text then
      Memo1.Lines.Add(LMessage.Role + ': ' +
        TsgcAI_MCP_Response_PromptsGet_Result_Message_Text(LMessage).Content.Text)
    else
      Memo1.Lines.Add(LMessage.Role + ': ' + LMessage.Write);
  end;
end;
```

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.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MCP.OnMCPListResources := MCPListResources;
  MCP.OnMCPResponseResource := MCPResponseResource;
end;
procedure TForm1.LoadResources;
begin
  MCP.ListResources;
end;
procedure TForm1.MCPListResources(Sender: TObject;
  const ARequest: TsgcAI_MCP_Request_ResourcesList;
  const AResponse: TsgcAI_MCP_Response_ResourcesList);
var
  LIndex: Integer;
  LResource: TsgcAI_MCP_Resource;
begin
  for LIndex := 0 to AResponse.Resources.Count - 1 do
  begin
    LResource := TsgcAI_MCP_Resource(AResponse.Resources.Item[LIndex]);
    Memo1.Lines.Add(Format('%s -> %s', [LResource.Name, LResource.Uri]));
  end;
end;
procedure TForm1.FetchResource(const AUri: string);
begin
  MCP.RequestResource(AUri);
end;
procedure TForm1.MCPResponseResource(Sender: TObject;
  const ARequest: TsgcAI_MCP_Request_ResourcesRead;
  const AResponse: TsgcAI_MCP_Response_ResourcesRead);
var
  LIndex: Integer;
  LContent: TsgcAI_MCP_Response_Resources_ReadResult_Content;
begin
  for LIndex := 0 to AResponse.Result.Contents.Count - 1 do
  begin
    LContent := TsgcAI_MCP_Response_Resources_ReadResult_Content(AResponse.Result.Contents.Item[LIndex]);
```

```
    if LContent.Text <> '' then
      Memo1.Lines.Add(LContent.Uri + ': ' + LContent.Text)
    else
      Memo1.Lines.Add(LContent.Uri + ' (binary, mime=' + LContent.MimeType + ')');
  end;
end;
```

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.

Sample code

The following snippets publish two roots when the server asks for them and log the JSON payload that will be sent back.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MCP.OnMCPListRoots := MCPListRoots;
end;
procedure TForm1.MCPListRoots(Sender: TObject;
  const ARequest: TsgcAI_MCP_Request_RootsList;
  const AResponse: TsgcAI_MCP_Response_RootsList);
begin
  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);
end;
```

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.

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.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MCP.OnMCPSamplingCreateMessage := MCPSamplingCreateMessage;
end;
procedure TForm1.MCPSamplingCreateMessage(Sender: TObject;
  const ARequest: TsgcAI_MCP_Request_SamplingCreateMessage;
  const AResponse: TsgcAI_MCP_Response_SamplingCreateMessage);
var
  LText: TsgcAI_MCP_Response_Result_Content_Text;
begin
  MemoLog.Lines.Add(Format('Sampling request with %d messages', [ARequest.Params.Messages.Count]));
  AResponse.Role := 'assistant';
  LText := TsgcAI_MCP_Response_Result_Content_Text.Create;
  LText.Text := 'The capital of France is Paris.';
  AResponse.Content := LText;
  AResponse.Model := 'claude-3-sonnet-20240307';
  AResponse.StopReason := 'endTurn';
  MemoLog.Lines.Add('Sampling response: ' + AResponse.Write);
end;
```

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.

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.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MCP.OnMCPElicitationCreate := MCPElicitationCreate;
end;
procedure TForm1.MCPElicitationCreate(Sender: TObject;
  const ARequest: TsgcAI_MCP_Request_ElicitationCreate;
  const AResponse: TsgcAI_MCP_Response_ElicitationCreate);
var
  LGitHub: string;
begin
  MemoLog.Lines.Add('Elicitation prompt: ' + ARequest.Params.Message);
  LGitHub := InputBox('GitHub', 'Provide your GitHub username', '');
  if LGitHub <> '' then
  begin
    AResponse.Action := 'accept';
    AResponse.Content.Clear;
    AResponse.Content.AddPair('name', LGitHub);
  end
  else
    AResponse.Action := 'decline';
  MemoLog.Lines.Add('Elicitation response: ' + AResponse.Write);
end;
```

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.

```
OpenAI := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
OpenAI := TsgcHTTP_OpenAI_JSON.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

oRequest := TsgcOpenAIClass_Request_Completion.Create;
Try
  oRequest.Model := 'text-davinci-003';
  oRequest.Prompt := 'Say this is a test';
  oRequest.Temperature := 1;
  oRequest.N := 2;
  oResponse := OpenAI.CreateCompletion(oRequest);

  if Length(oResponse.Choices) > 0 then
    WriteLn(oResponse.Choices[0].Text);
Finally
  oRequest.Free;
  oResponse.Free;
End;
```

OpenAI | Chat

Given a chat conversation, the model will return a chat completion response.

Simple Example

Interact with ChatGPT by sending a Hello message.

```
OpenAI := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
OpenAI := TsgcHTTP_OpenAI_JSON.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

oRequest := TsgcOpenAIClass_Request_ChatCompletion.Create;
Try
  oRequest.Model := 'gpt-3.5-turbo';
  oMessage := TsgcOpenAIClass_Request_Completion_Message.Create;
  oMessage.Role := 'user';
  oMessage.Content := 'Hello!';
  oMessages := oRequest.Messages;
  SetLength(oMessages, 1);
  oMessages[0] := oMessage;
  oRequest.Messages := oMessages;
  oRequest.Temperature := 1;
  oRequest.N := 2;
  oResponse := OpenAI.CreateChatCompletion(oRequest);

  if Length(oResponse.Choices) > 0 then
    WriteLn(oResponse.Choices[0]._Message.Content);
Finally
  oRequest.Free;
  oResponse.Free;
End;
```

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.

```
OpenAI := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
OpenAI := TsgcHTTP_OpenAI_JSON.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

oRequest := TsgcOpenAIClass_Request_Edit.Create;
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;
  oResponse := OpenAI.CreateEdit(oRequest);

  if Length(oResponse.Choices) > 0 then
    WriteLn(oResponse.Choices[0].Text);
Finally
  oRequest.Free;
  oResponse.Free;
End;
```

OpenAI | Audio

Create Transcription

Transcribes audio into the input language.

```
OpenAI := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

WriteLn(OpenAI._CreateTranscriptionFromFile('whisper-1', 'c:\media\audio.mp3'));
```

Create Translation

Translates audio to English.

```
OpenAI := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

WriteLn(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

```
OpenAI := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

WriteLn(OpenAI._CreateModeration('I want to kill them.'));
```

Advanced Example

Moderate the following text choosing the model.

```
OpenAI := TsgcHTTP_OpenAI_JSON.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

oRequest := TsgcOpenAIClass_Request_Moderation.Create;
Try
  oRequest.Model := 'text-moderation-latest';
  oRequest.Input := 'I want to kill them.';
  oResponse := OpenAI.CreateModeration(oRequest);

  if Length(oResponse.results) > 0 then
    WriteLn(oResponse.results[0].flagged);
Finally
  oRequest.Free;
  oResponse.Free;
End;
```

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.

```
WSClient := TsgcWebSocketClient.Create(nil);
oAudio := TsgcAudioRecorderWave.Create(nil);
OpenAI := TsgcWSAPI_OpenAI.Create(nil);
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';
procedure OnOpenAIAudioTranscriptionCompleted(Sender: TObject; const aItem: TsgcWSOpenAIConversation_Item_Complet
begin
  Log('#transcription_completed: ' + aItem.Transcript);
end;
```

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.

```
OpenAI := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';
WriteLn(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 := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

oStream := TFileStream.Create('output.mp3', fmCreate);
Try
  OpenAI._CreateSpeech('tts-1', 'Hello, how are you?', 'alloy', oStream);
Finally
  oStream.Free;
End;
```

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.

```
OpenAI := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

// Create a fine-tuning job
WriteLn(OpenAI._CreateFineTuningJob('gpt-4o-mini-2024-07-18', 'file-abc123'));

// List fine-tuning jobs
WriteLn(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.

```
OpenAI := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

// Create a batch
WriteLn(OpenAI._CreateBatch('file-abc123', '/v1/chat/completions'));

// List batches
WriteLn(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.

```
OpenAI := TsgcHTTP_API_OpenAI.Create(nil);
OpenAI.OpenAIOptions.ApiKey := 'API_KEY';

// Create an upload
vUploadResponse := OpenAI._CreateUpload('training_data.jsonl', 'fine-tune',
  2147483648, 'application/jsonl');

// Add a part
vPartResponse := OpenAI._AddUploadPart('upload_abc123', 'part1.jsonl');

// Complete the upload
WriteLn(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
oAssistant := TsgcAIOpenAIAssistant.Create(nil);
// ... 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
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.

```
procedure SendMessage(const oAssistant: TsgcAIOpenAIAssistant; const oThread: TsgcAIClass_Thread; const aMess
var
  i: Integer;
  oMessage: TsgcOpenAIClass_Message;
  oMessages: TsgcOpenAIClass_Response_List_Messages;
  oRun: TsgcOpenAIClass_Run;
begin
  DoLog('[user]: ' + aMessage);
  oMessage := oAssistant.CreateMessageText(oThread.Id, aMessage);
  if Assigned(oMessage) then
  begin
    oRun := oAssistant.CreateRunAndWait(oThread.Id);
    if Assigned(oRun) then
    begin
      oMessages := oAssistant.GetMessages(oThread.Id, oRun.Id);
      if Assigned(oMessages) and (Length(oMessages.Messages) > 0) then
      begin
        for i := 0 to Length(oMessages.Messages) - 1 do
          DoLog('[assistant]: ' + DoFormatResponse(oMessages.Messages[i].ContentText + #13#10));
      end;
    end;
  end;
end;
```

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.

```
Assistant := TsgcAIOpenAIAssistant.Create(nil);
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 question.'
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.

```
procedure UploadFile();
var
  oDialog: TOpenDialog;
begin
  oDialog := TOpenDialog.Create(nil);
  Try
    if oDialog.Execute then
    begin
      Screen.Cursor := crHourGlass;
      Try
        Assistant.UploadVectorStoreFile('sgcVectorStore', oDialog.FileName);
      Finally
        Screen.Cursor := crDefault;
      End;
    end;
  Finally
    oDialog.Free;
  End;
end;
```

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.

```
procedure SendMessage()
var
  i: Integer;
  oMessage: TsgcOpenAIClass_Message;
  oMessages: TsgcOpenAIClass_Response_List_Messages;
  oRun: TsgcOpenAIClass_Run;
begin
  DoLog('[user]: ' + memoMessage.Lines.Text);
  Screen.Cursor := crHourGlass;
  Try
    oMessage := Assistant.CreateMessageText('thread_id', 'Create a WebSocket Client that connects to eSeGeCe WebS
    if Assigned(oMessage) then
    begin
      oRun := Assistant.CreateRunAndWait('thread_id');
      if Assigned(oRun) then
      begin
        oMessages := Assistant.GetMessages('thread_id', oRun.Id);
        if Assigned(oMessages) and (Length(oMessages.Messages) > 0) then
        begin
          memoMessage.Lines.Text := '';
          for i := 0 to Length(oMessages.Messages) - 1 do
            DoLog('[assistant]: ' + DoFormatResponse(oMessages.Messages[i]
              .ContentText + #13#10));
        end;
      end;
    end;
  Finally
    Screen.Cursor := crDefault;
  End;
end;
```

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
oAssistant := TsgcAIOpenAIAssistant.Create(nil);
// ... 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 mat
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.

```
procedure SendMessage(const oAssistant: TsgcAIOpenAIAssistant; const oThread: TsgcAIClass_Thread; const aMessage:
var
  i: Integer;
```

```
  oMessage: TsgcOpenAIClass_Message;
  oMessages: TsgcOpenAIClass_Response_List_Messages;
  oRun: TsgcOpenAIClass_Run;
begin
  DoLog('[user]: ' + aMessage);
  oMessage := oAssistant.CreateMessageText(oThread.Id, aMessage);
  if Assigned(oMessage) then
    oRun := oAssistant.CreateRun(oThread.Id, True);
end;
```

Step 4: Handle the Response

Use the event OnStreamMessageDelta to read the server-sent stream message.

```
procedure OnStreamMessageDelta(Sender: TObject; const aMessageDelta: TsgcOpenAIClass_MessageDelta; const aRaw: st
var
  i: Integer;
  vResponse: string;
  vType: string;
begin
  for i := Low(aMessageDelta.Content) to High(aMessageDelta.Content) do
  begin
    vType := aMessageDelta.Content[i]._Type;
    if vType = 'text' then
    begin
      vResponse := TsgcOpenAIClass_MessageDeltaContent_Text
        (aMessageDelta.Content[i]).Value;
    end;
  end;
end;
```

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.

```
Assistant := TsgcAIOpenAIAssistant.Create(nil);
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 ques
Assistant.AssistantOptions.Model := 'gpt-4o';
Assistant.AssistantOptions.Tools.Functions.Enabled := False;
Assistant.AssistantOptions.Tools.Functions.Functions.Text := '[{"type":"function","function":{"name":"get_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.

```
procedure SendMessage()
var
  i: Integer;
  oMessage: TsgcOpenAIClass_Message;
  oMessages: TsgcOpenAIClass_Response_List_Messages;
  oRun: TsgcOpenAIClass_Run;
begin
  DoLog('[user]: ' + memoMessage.Lines.Text);
  Screen.Cursor := crHourGlass;
  Try
    oMessage := Assistant.CreateMessageText('thread_id', 'What is the weather in San Francisco today and the like
    if Assigned(oMessage) then
    begin
      oRun := Assistant.CreateRunAndWait('thread_id');
      if Assigned(oRun) then
      begin
        oMessages := Assistant.GetMessages('thread_id', oRun.Id);
        if Assigned(oMessages) and (Length(oMessages.Messages) > 0) then
        begin
          memoMessage.Lines.Text := '';
          for i := 0 to Length(oMessages.Messages) - 1 do
            DoLog('[assistant]: ' + DoFormatResponse(oMessages.Messages[i]
              .ContentText + #13#10));
        end;
      end;
    end;
  Finally
    Screen.Cursor := crDefault;
  End;
end;
```

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.

```
procedure TFRMOpenAIAssistant.AssistantFunctionCall(Sender: TObject;
  const aRequest: TsgcOpenAIClass_ToolCall;
  const aResponse: TsgcHTTPOpenAI_ToolCall_Response);
begin
  if aRequest._Function._Name = 'get_current_temperature' then
    aResponse.Output := 30
  else if aRequest._Function._Name = 'get_rain_probability' then
    aResponse.Output := 10;
end;
```

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

Reference

- Properties — 5 properties available on the component.
- Methods — 4 public methods.
- Events — 3 events you can handle.

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.

```
procedure ConvertFileToVector;
var
  oDialog: TOpenDialog;
  oEmbeddings: TsgcAIOpenAIEmbeddings;
  oFile: TsgcAIDatabaseVectorFile;
begin
  oDialog := TOpenDialog.Create(nil);
  Try
    oDialog.Filter := 'TXT Files|*.txt';
    if oDialog.Execute then
    begin
      oEmbeddings := TsgcAIOpenAIEmbeddings.Create(nil);
      Try
        oFile := TsgcAIDatabaseVectorFile.Create(nil);
        Try
          oEmbeddings.Database := oFile;
          oEmbeddings.OpenAIOptions.ApiKey := '<your api key>';
          oEmbeddings.CreateEmbeddingsFromFile(oDialog.FileName);
        Finally
          oFile.Free;
        End;
      Finally
        oEmbeddings.Free;
      End;
    end;
  Finally
    FreeAndNil(oDialog);
  End;
end;
```

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.

```
procedure AskToChatGPT(const aQuestion: string);
var
  oChatBot: TsgcAIOpenAIChatBot;
  oEmbeddings: TsgcAIOpenAIEmbeddings;
  oFile: TsgcAIDatabaseVectorFile;
  vContext: string;
begin
  oChatBot := TsgcAIOpenAIChatBot.Create(nil);
  Try
    oChatBot.OpenAIOptions.ApiKey := '<your api key>';
    oEmbeddings := TsgcAIOpenAIEmbeddings.Create(nil);
    Try
      oChatBot.Embeddings := oEmbeddings;
      oFile := TsgcAIDatabaseVectorFile.Create(nil);
      Try
        oEmbeddings.Database := oFile;
        vContext := oChatBot.GetEmbedding(aQuestion);
        oChatBot.ChatAsUser('Answer the question based on the context below.\n\nContext:\n' +
          vContext + '\nQuestion:' + aQuestion + '\nAnswer:');
      Finally
        oFile.Free;
      End;
    Finally
      oEmbeddings.Free;
    End;
  Finally
    FreeAndNil(oDialog);
  End;
end;
```

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".

```
procedure UpsertPinecone(const aIndexName, aProjectId: string; const aVector: Array of Double);
var
  oPinecone: TsgcHTTP_API_Pinecone;
  oParams: TsgcHTTPPineconeVectorUpserts;
  oVectors: TsgcArrayOfVectorUpsert;
begin
  oPinecone := TsgcHTTP_API_Pinecone.Create(nil);
  Try
    oPinecone.PineconeOptions.API := 'your-api-key';
    oParams := TsgcHTTPPineconeVectorUpserts.Create;
    Try
      SetLength(oVectors, 1);
      oVectors[0] := TsgcHTTPPineconeVectorUpsert.Create;
      oVectors[0].Id := 'id1';
      oVectors[0].Values := aVector;
      oParams.Vectors := oVectors;
      Pinecone.VectorsUpsert(aIndexName, aProjectId, oParams);
    Finally
      oParams.Free;
    End;
  Finally
    oPinecone.Free;
  End;
end;
```

Example QUERY

Find below an example of QUERY a single vector.

```
procedure QueryPinecone(const aIndexName, aProjectId: string; const aVector: Array of Double);
var
  oParams: TsgcHTTPPineconeVectorQuery;
begin
  oParams := TsgcHTTPPineconeVectorQuery.Create;
  Try
    oParams.Vector := aVector;
    Pinecone.VectorsQuery(aIndexName, aProjectId, oParams);
  Finally
    oParams.Free;
  End;
end;
```

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).

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
WriteLn(Anthropic._CreateMessage('claude-sonnet-4-20250514', 'Hello!'));
```

System Prompt Example

Send a message with a system prompt to control Claude's behavior.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

oRequest := TsgcAnthropicClass_Request_Messages.Create;
Try
  oRequest.Model := 'claude-sonnet-4-20250514';
  oRequest.MaxTokens := 1024;
  oRequest.System := 'You are a helpful assistant.';
  oRequest.Temperature := 0.7;

  oMessage := TsgcAnthropicClass_Request_Message.Create;
  oMessage.Role := 'user';
  oMessage.Content := 'Hello!';
  oMessages := oRequest.Messages;
  SetLength(oMessages, 1);
  oMessages[0] := oMessage;
  oRequest.Messages := oMessages;

  oResponse := Anthropic.CreateMessage(oRequest);
  Try
    if Length(oResponse.Content) > 0 then
      WriteLn(oResponse.Content[0].Text);
  Finally
    oResponse.Free;
  End;
Finally
  oMessage.Free;
  oRequest.Free;
End;
```

Streaming Example

Use Server-Sent Events (SSE) to stream the response in real-time. Assign the OnHTTPAPISSE event handler to receive streaming events.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
```

```
Anthropic.OnHTTPAPISSE := OnSSEEvent;
Anthropic._CreateMessageStream('claude-sonnet-4-20250514', 'Tell me a story.');

procedure TForm1.OnSSEEvent(Sender: TObject; const aEvent, aData: string;
  var Cancel: Boolean);
begin
  // aEvent contains the event type (e.g. content_block_delta)
  // aData contains the JSON data for this event
  Memo1.Lines.Add(aData);
end;
```

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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

// Load image and encode to base64
oStream := TFileStream.Create('photo.png', fmOpenRead);
Try
  oBytes := TBytesStream.Create;
  Try
    oBytes.CopyFrom(oStream, 0);
    vBase64 := EncodeBase64(oBytes.Memory, oBytes.Size);
  Finally
    oBytes.Free;
  End;
Finally
  oStream.Free;
End;

WriteLn(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.

```
oRequest := TsgcAnthropicClass_Request_Messages.Create;
Try
  oRequest.Model := 'claude-sonnet-4-20250514';
  oRequest.MaxTokens := 4096;

  oMessage := TsgcAnthropicClass_Request_Message.Create;
  oMessage.Role := 'user';

  // Image content block
  oImageBlock := TsgcAnthropicClass_Request_Content_Block.Create;
  oImageBlock.ContentType := 'image';
  oImageBlock.MediaType := 'image/jpeg';
  oImageBlock.Data := vBase64;

  // Text content block
  oTextBlock := TsgcAnthropicClass_Request_Content_Block.Create;
  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;

  oResponse := Anthropic.CreateMessage(oRequest);
  Try
    if Length(oResponse.Content) > 0 then
```

```
      WriteLn(oResponse.Content[0].Text);
  Finally
    oResponse.Free;
  End;
Finally
  oImageBlock.Free;
  oTextBlock.Free;
  oMessage.Free;
  oRequest.Free;
End;
```

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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

// Step 1: Create request with tools
oRequest := TsgcAnthropicClass_Request_Messages.Create;
Try
  oRequest.Model := 'claude-sonnet-4-20250514';
  oRequest.MaxTokens := 4096;

  // Define user message
  oMessage := TsgcAnthropicClass_Request_Message.Create;
  oMessage.Role := 'user';
  oMessage.Content := 'What is the weather in San Francisco?';
  SetLength(oMessages, 1);
  oMessages[0] := oMessage;
  oRequest.Messages := oMessages;

  // Define tool
  oTool := TsgcAnthropicClass_Request_Tool.Create;
  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
  oResponse := Anthropic.CreateMessage(oRequest);
  Try
    // Step 3: Check if Claude wants to use a tool
    if oResponse.StopReason = 'tool_use' then
    begin
      // Find the tool_use content block
      for i := 0 to Length(oResponse.Content) - 1 do
      begin
        if oResponse.Content[i].ContentType = 'tool_use' then
        begin
          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;
        end;
      end;

      // Step 5: Send tool result back
      // Build new message array with assistant response + tool result
      // ... (continue the conversation with tool_result content block)
    end;
  Finally
    oResponse.Free;
```

```
  End;
Finally
  oMessage.Free;
  oTool.Free;
  oRequest.Free;
End;
```

Anthropic | Models

List and retrieve information about available Claude models.

List Models

Lists all available Claude models.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
WriteLn(Anthropic._GetModels);
```

Get Model

Retrieves information about a specific model.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
WriteLn(Anthropic._GetModel('claude-sonnet-4-20250514'));
```

Typed Response

Use the typed response class for structured access to model data.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

oModels := Anthropic.GetModels;
Try
  for i := 0 to Length(oModels.Data) - 1 do
    WriteLn(oModels.Data[i].Id + ' - ' + oModels.Data[i].DisplayName);
Finally
  oModels.Free;
End;
```

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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
WriteLn(Anthropic._ListBatches);
```

Get Batch

Retrieves information about a specific batch.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
WriteLn(Anthropic._GetBatch('batch_id_here'));
```

Cancel Batch

Cancels a batch that is still processing.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
WriteLn(Anthropic._CancelBatch('batch_id_here'));
```

Get Batch Results

Retrieves the results of a completed batch.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
WriteLn(Anthropic._GetBatchResults('batch_id_here'));
```

Typed Response

Use the typed response class for structured access to batch data.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

oBatches := Anthropic.ListBatches;
Try
  for i := 0 to Length(oBatches.Batches) - 1 do
    WriteLn(oBatches.Batches[i].Id + ' - ' + oBatches.Batches[i].ProcessingStatus);
Finally
  oBatches.Free;
End;
```

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).

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

oRequest := TsgcAnthropicClass_Request_Messages.Create;
Try
  oRequest.Model := 'claude-sonnet-4-20250514';
  oRequest.MaxTokens := 16384;
  oRequest.ThinkingType := 'enabled';
  oRequest.ThinkingBudgetTokens := 10000;

  oMessage := TsgcAnthropicClass_Request_Message.Create;
  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;

  oResponse := Anthropic.CreateMessage(oRequest);
  Try
    for i := 0 to Length(oResponse.Content) - 1 do
    begin
      if oResponse.Content[i].ContentType = 'thinking' then
        WriteLn('Thinking: ' + oResponse.Content[i].Thinking)
      else if oResponse.Content[i].ContentType = 'text' then
        WriteLn('Response: ' + oResponse.Content[i].Text);
    end;
  Finally
    oResponse.Free;
  End;
Finally
  oMessage.Free;
  oRequest.Free;
End;
```

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
oBlock := TsgcAnthropicClass_Request_Content_Block.Create;
oBlock.ContentType := 'thinking';
oBlock.Text := oPrevThinkingBlock.Thinking;   // thinking text
oBlock.Signature := oPrevThinkingBlock.Signature; // signature string
```

```
// Pass back redacted thinking blocks
oBlock := TsgcAnthropicClass_Request_Content_Block.Create;
oBlock.ContentType := 'redacted_thinking';
oBlock.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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

// Load PDF file and encode to base64
vBase64 := sgcBase64Encode(LoadFileToBytes('document.pdf'));
WriteLn(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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

oRequest := TsgcAnthropicClass_Request_Messages.Create;
Try
  oRequest.Model := 'claude-sonnet-4-20250514';
  oRequest.MaxTokens := 4096;

  // Create document content block
  oDocBlock := TsgcAnthropicClass_Request_Content_Block.Create;
  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
  oTextBlock := TsgcAnthropicClass_Request_Content_Block.Create;
  oTextBlock.ContentType := 'text';
  oTextBlock.Text := 'What are the key findings?';

  // Build message with content blocks
  oMessage := TsgcAnthropicClass_Request_Message.Create;
  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;

  oResponse := Anthropic.CreateMessage(oRequest);
  Try
    if Length(oResponse.Content) > 0 then
      WriteLn(oResponse.Content[0].Text);
  Finally
    oResponse.Free;
  End;
Finally
  oDocBlock.Free;
  oTextBlock.Free;
  oMessage.Free;
  oRequest.Free;
End;
```

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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

oRequest := TsgcAnthropicClass_Request_Messages.Create;
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

  oMessage := TsgcAnthropicClass_Request_Message.Create;
  oMessage.Role := 'user';
  oMessage.Content := 'Analyze this contract clause.';
  oMessages := oRequest.Messages;
  SetLength(oMessages, 1);
  oMessages[0] := oMessage;
  oRequest.Messages := oMessages;

  oResponse := Anthropic.CreateMessage(oRequest);
  Try
    // Check cache usage in response
    WriteLn('Cache created: ' +
      IntToStr(oResponse.Usage.CacheCreationInputTokens));
    WriteLn('Cache read: ' +
      IntToStr(oResponse.Usage.CacheReadInputTokens));
    WriteLn(oResponse.Content[0].Text);
  Finally
    oResponse.Free;
  End;
Finally
  oMessage.Free;
  oRequest.Free;
End;
```

Content Block Caching

Cache specific content blocks (e.g. large documents or context) by setting CacheControl to 'ephemeral' on individual content blocks.

```
oBlock := TsgcAnthropicClass_Request_Content_Block.Create;
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.

```
oTool := TsgcAnthropicClass_Request_Tool.Create;
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.

```
oDocBlock := TsgcAnthropicClass_Request_Content_Block.Create;
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.

```
oResponse := Anthropic.CreateMessage(oRequest);
Try
  for i := 0 to Length(oResponse.Content) - 1 do
  begin
    if oResponse.Content[i].ContentType = 'text' then
    begin
      WriteLn(oResponse.Content[i].Text);

      // Process citations
      for j := 0 to Length(oResponse.Content[i].Citations) - 1 do
      begin
        oCitation := oResponse.Content[i].Citations[j];
        WriteLn(Format('  Citation [%s]: "%s"',
          [oCitation.CitationType, oCitation.CitedText]));

        if oCitation.CitationType = 'page_location' then
          WriteLn(Format('    Pages %d-%d of "%s"',
            [oCitation.StartPageNumber, oCitation.EndPageNumber,
             oCitation.DocumentTitle]));
      end;
    end;
  end;
Finally
  oResponse.Free;
End;
```

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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

oRequest := TsgcAnthropicClass_Request_Messages.Create;
Try
  oRequest.Model := 'claude-sonnet-4-20250514';
  oRequest.MaxTokens := 4096;

  // Add web search tool
  oTool := TsgcAnthropicClass_Request_Tool.Create;
  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;

  oMessage := TsgcAnthropicClass_Request_Message.Create;
  oMessage.Role := 'user';
  oMessage.Content := 'Find the current price of Bitcoin.';
  oMessages := oRequest.Messages;
  SetLength(oMessages, 1);
  oMessages[0] := oMessage;
  oRequest.Messages := oMessages;

  oResponse := Anthropic.CreateMessage(oRequest);
  Try
    for i := 0 to Length(oResponse.Content) - 1 do
    begin
      if oResponse.Content[i].ContentType = 'text' then
        WriteLn(oResponse.Content[i].Text);
    end;
  Finally
    oResponse.Free;
  End;
Finally
  oTool.Free;
  oMessage.Free;
  oRequest.Free;
End;
```

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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

vSchema := '{"type":"object","properties":{"name":{"type":"string"},' +
  '"age":{"type":"integer"}},"required":["name","age"],' +
  '"additionalProperties":false}';

WriteLn(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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';

oRequest := TsgcAnthropicClass_Request_Messages.Create;
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';

  oMessage := TsgcAnthropicClass_Request_Message.Create;
  oMessage.Role := 'user';
  oMessage.Content := 'Analyze the sentiment: I love this product!';
  oMessages := oRequest.Messages;
  SetLength(oMessages, 1);
  oMessages[0] := oMessage;
  oRequest.Messages := oMessages;

  oResponse := Anthropic.CreateMessage(oRequest);
  Try
    for i := 0 to Length(oResponse.Content) - 1 do
    begin
      if oResponse.Content[i].ContentType = 'text' then
        WriteLn(oResponse.Content[i].Text);
    end;
  Finally
    oResponse.Free;
  End;
Finally
  oMessage.Free;
  oRequest.Free;
End;
```

Strict Tool Use

Enable strict mode on tool definitions to guarantee tool inputs conform exactly to the input_schema.

```
oTool := TsgcAnthropicClass_Request_Tool.Create;
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

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
Anthropic.AnthropicOptions.BetaHeaders := 'files-api-2025-04-14';

oFile := Anthropic.UploadFile('C:\documents\report.pdf');
Try
  WriteLn('File ID: ' + oFile.Id);
  WriteLn('Filename: ' + oFile.Filename);
  WriteLn('Size: ' + IntToStr(oFile.SizeBytes) + ' bytes');
  WriteLn('MIME Type: ' + oFile.MimeType);
Finally
  oFile.Free;
End;
```

List Files

```
oList := Anthropic.ListFiles(50);
Try
  for i := 0 to Length(oList.Data) - 1 do
    WriteLn(Format('%s - %s (%d bytes)',
      [oList.Data[i].Id, oList.Data[i].Filename, oList.Data[i].SizeBytes]));
  WriteLn('Has more: ' + BoolToStr(oList.HasMore, True));
Finally
  oList.Free;
End;
```

Use File in Messages

Reference uploaded files using the document content block with SourceType set to 'file'.

```
oDocBlock := TsgcAnthropicClass_Request_Content_Block.Create;
oDocBlock.ContentType := 'document';
oDocBlock.SourceType := 'file';
oDocBlock.FileId := 'file_abc123';  // ID from UploadFile

oTextBlock := TsgcAnthropicClass_Request_Content_Block.Create;
oTextBlock.ContentType := 'text';
oTextBlock.Text := 'Summarize this document.';
```

Delete a File

```
oDeleted := Anthropic.DeleteFile('file_abc123');
Try
  WriteLn('Deleted: ' + oDeleted.Id);
Finally
  oDeleted.Free;
End;
```

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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
Anthropic.AnthropicOptions.BetaHeaders := 'mcp-client-2025-11-20';

WriteLn(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.

```
Anthropic := TsgcHTTP_API_Anthropic.Create(nil);
Anthropic.AnthropicOptions.ApiKey := 'API_KEY';
Anthropic.AnthropicOptions.BetaHeaders := 'mcp-client-2025-11-20';

oRequest := TsgcAnthropicClass_Request_Messages.Create;
Try
  oRequest.Model := 'claude-sonnet-4-20250514';
  oRequest.MaxTokens := 4096;

  // Configure MCP server
  oServer := TsgcAnthropicClass_Request_MCPServer.Create;
  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
  oTool := TsgcAnthropicClass_Request_Tool.Create;
  oTool.ToolType := 'mcp_toolset';
  oTool.MCPServerName := 'my-server';
  oTools := oRequest.Tools;
  SetLength(oTools, 1);
  oTools[0] := oTool;
  oRequest.Tools := oTools;

  // Add user message
  oMessage := TsgcAnthropicClass_Request_Message.Create;
  oMessage.Role := 'user';
  oMessage.Content := 'Search for recent news about AI.';
  oMessages := oRequest.Messages;
  SetLength(oMessages, 1);
  oMessages[0] := oMessage;
  oRequest.Messages := oMessages;

  oResponse := Anthropic.CreateMessage(oRequest);
  Try
    for i := 0 to Length(oResponse.Content) - 1 do
    begin
      if oResponse.Content[i].ContentType = 'text' then
        WriteLn(oResponse.Content[i].Text)
      else if oResponse.Content[i].ContentType = 'mcp_tool_use' then
        WriteLn(Format('MCP tool call: %s on %s',
          [oResponse.Content[i].Name, oResponse.Content[i].ServerName]));
    end;
```

```
  Finally
    oResponse.Free;
  End;
Finally
  oServer.Free;
  oTool.Free;
  oMessage.Free;
  oRequest.Free;
End;
```

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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
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.

```
DeepSeek := TsgcHTTP_API_DeepSeek.Create(nil);
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.

```
DeepSeek := TsgcHTTP_API_DeepSeek.Create(nil);
DeepSeek.DeepSeekOptions.ApiKey := 'API_KEY';
WriteLn(DeepSeek._CreateMessage('deepseek-chat', 'Hello!'));
```

System Prompt Example

Send a message with a system prompt to control DeepSeek's behavior.

```
DeepSeek := TsgcHTTP_API_DeepSeek.Create(nil);
DeepSeek.DeepSeekOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
DeepSeek := TsgcHTTP_API_DeepSeek.Create(nil);
DeepSeek.DeepSeekOptions.ApiKey := 'API_KEY';
DeepSeek.OnHTTPAPISSE := OnSSEEvent;
DeepSeek._CreateMessageStream('deepseek-chat', 'Tell me a story.');

procedure TForm1.OnSSEEvent(Sender: TObject; const aEvent, aData: string;
  var Cancel: Boolean);
begin
  // aEvent contains the event type
  // aData contains the JSON data for this event
  Memo1.Lines.Add(aData);
end;
```

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.

```
DeepSeek := TsgcHTTP_API_DeepSeek.Create(nil);
DeepSeek.DeepSeekOptions.ApiKey := 'API_KEY';

// Load image and encode to base64
oStream := TFileStream.Create('photo.jpg', fmOpenRead);
Try
  oBytes := TBytesStream.Create;
  Try
    oBytes.CopyFrom(oStream, 0);
    vImageBase64 := EncodeBase64(oBytes.Memory, oBytes.Size);
  Finally
    oBytes.Free;
  End;
Finally
  oStream.Free;
End;

WriteLn(DeepSeek._CreateVisionMessage('deepseek-chat',
  'Describe this image', vImageBase64, 'image/jpeg'));
```

DeepSeek | Models

List the available DeepSeek models.

List Models

Lists all available DeepSeek models.

```
DeepSeek := TsgcHTTP_API_DeepSeek.Create(nil);
DeepSeek.DeepSeekOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';
WriteLn(Gemini._CreateContent('gemini-2.0-flash', 'Hello!'));
```

System Instruction Example

Send a message with a system instruction to control Gemini's behavior.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';
Gemini.OnHTTPAPISSE := OnSSEEvent;
Gemini._CreateContentStream('gemini-2.0-flash', 'Tell me a story.');

procedure TForm1.OnSSEEvent(Sender: TObject; const aEvent, aData: string;
  var Cancel: Boolean);
begin
  Memo1.Lines.Add(aData);
end;
```

Advanced Example

Use the typed request/response classes for full control over content generation parameters.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';

oRequest := TsgcGeminiClass_Request_GenerateContent.Create;
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] := TsgcGeminiClass_Request_Content.Create;
  oContents[0].Role := 'user';
  SetLength(oParts, 1);
  oParts[0] := TsgcGeminiClass_Request_Part.Create;
  oParts[0].Text := 'Write a haiku about programming.';
  oContents[0].Parts := oParts;
  oRequest.Contents := oContents;

  oResponse := Gemini.CreateContent(oRequest);
  Try
```

```
    if Length(oResponse.Candidates) > 0 then
      if Length(oResponse.Candidates[0].Parts) > 0 then
        WriteLn(oResponse.Candidates[0].Parts[0].Text);
  Finally
    oResponse.Free;
  End;
Finally
  oParts[0].Free;
  oContents[0].Free;
  oRequest.Free;
End;
```

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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';

// Load image and encode to base64
oStream := TFileStream.Create('photo.jpg', fmOpenRead);
Try
  oBytes := TBytesStream.Create;
  Try
    oBytes.CopyFrom(oStream, 0);
    vImageBase64 := EncodeBase64(oBytes.Memory, oBytes.Size);
  Finally
    oBytes.Free;
  End;
Finally
  oStream.Free;
End;

WriteLn(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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';
WriteLn(Gemini._GetModels);
```

Get Model

Get details for a specific model.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';
WriteLn(Gemini._GetModel('gemini-2.0-flash'));
```

Advanced Example

Use the typed response classes for full control over model data.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';

oModels := Gemini.GetModels;
Try
  for i := 0 to Length(oModels.Models) - 1 do
    WriteLn(oModels.Models[i].DisplayName + ' - ' + oModels.Models[i].Description);
Finally
  oModels.Free;
End;
```

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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';

vSchema := '{"type":"object","properties":{"name":{"type":"string"},' +
  '"age":{"type":"integer"}},"required":["name","age"]}';

WriteLn(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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';

oRequest := TsgcGeminiClass_Request_GenerateContent.Create;
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] := TsgcGeminiClass_Request_Content.Create;
  oContents[0].Role := 'user';
  SetLength(oParts, 1);
  oParts[0] := TsgcGeminiClass_Request_Part.Create;
  oParts[0].Text := 'Analyze the sentiment: I love this product!';
  oContents[0].Parts := oParts;
  oRequest.Contents := oContents;

  oResponse := Gemini.CreateContent(oRequest);
  Try
    if Length(oResponse.Candidates) > 0 then
      if Length(oResponse.Candidates[0].Parts) > 0 then
        WriteLn(oResponse.Candidates[0].Parts[0].Text);
  Finally
    oResponse.Free;
  End;
Finally
  oParts[0].Free;
  oContents[0].Free;
  oRequest.Free;
End;
```

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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';

oRequest := TsgcGeminiClass_Request_CountTokens.Create;
Try
  oRequest.Model := 'gemini-2.0-flash';
  SetLength(oContents, 1);
  oContents[0] := TsgcGeminiClass_Request_Content.Create;
  oContents[0].Role := 'user';
  SetLength(oParts, 1);
  oParts[0] := TsgcGeminiClass_Request_Part.Create;
  oParts[0].Text := 'Explain quantum computing in simple terms.';
  oContents[0].Parts := oParts;
  oRequest.Contents := oContents;

  oResponse := Gemini.CountTokens(oRequest);
  Try
    WriteLn('Total tokens: ' + IntToStr(oResponse.TotalTokens));
  Finally
    oResponse.Free;
  End;
Finally
  oParts[0].Free;
  oContents[0].Free;
  oRequest.Free;
End;
```

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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';
WriteLn(Gemini._EmbedContent('text-embedding-004', 'Hello world'));
```

Advanced Example

Use the typed response classes for full control over embedding data.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';

oEmbedding := Gemini.EmbedContent('text-embedding-004', 'Hello world');
Try
  WriteLn('Dimensions: ' + IntToStr(Length(oEmbedding.Values)));
  for i := 0 to Length(oEmbedding.Values) - 1 do
    WriteLn(FloatToStr(oEmbedding.Values[i]));
Finally
  oEmbedding.Free;
End;
```

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.

```
Gemini := TsgcHTTP_API_Gemini.Create(nil);
Gemini.GeminiOptions.ApiKey := 'API_KEY';

// Step 1: Create request with function declarations
oRequest := TsgcGeminiClass_Request_GenerateContent.Create;
Try
  oRequest.Model := 'gemini-2.0-flash';
  oRequest.MaxOutputTokens := 4096;

  // Define user message
  SetLength(oContents, 1);
  oContents[0] := TsgcGeminiClass_Request_Content.Create;
  oContents[0].Role := 'user';
  SetLength(oParts, 1);
  oParts[0] := TsgcGeminiClass_Request_Part.Create;
  oParts[0].Text := 'What is the weather in San Francisco?';
  oContents[0].Parts := oParts;
  oRequest.Contents := oContents;

  // Define function
  SetLength(oFunctions, 1);
  oFunctions[0] := TsgcGeminiClass_Request_FunctionDeclaration.Create;
  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
  oResponse := Gemini.CreateContent(oRequest);
  Try
    // Step 3: Check for function call
    if Length(oResponse.Candidates) > 0 then
    begin
      if Length(oResponse.Candidates[0].Parts) > 0 then
      begin
        if oResponse.Candidates[0].Parts[0].FunctionCallName <> '' then
        begin
          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';
        end;
      end;
    end;
  Finally
    oResponse.Free;
  End;
Finally
  oParts[0].Free;
  oContents[0].Free;
  oFunctions[0].Free;
  oRequest.Free;
```

```
End;
```

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.

```
Ollama := TsgcHTTP_API_Ollama.Create(nil);
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.

```
Ollama := TsgcHTTP_API_Ollama.Create(nil);
Ollama.OllamaOptions.Host := 'http://localhost:11434';
WriteLn(Ollama._CreateMessage('llama3', 'Hello!'));
```

System Prompt Example

Send a message with a system prompt to control the model's behavior.

```
Ollama := TsgcHTTP_API_Ollama.Create(nil);
Ollama.OllamaOptions.Host := 'http://localhost:11434';
WriteLn(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.

```
Ollama := TsgcHTTP_API_Ollama.Create(nil);
Ollama.OllamaOptions.Host := 'http://localhost:11434';
Ollama.OnHTTPAPISSE := OnSSEEvent;
Ollama._CreateMessageStream('llama3', 'Tell me a story.');

procedure TForm1.OnSSEEvent(Sender: TObject; const aEvent, aData: string;
  var Cancel: Boolean);
begin
  // aEvent contains the event type
  // aData contains the JSON data for this event
  Memo1.Lines.Add(aData);
end;
```

Ollama | Models

Manage and query locally available Ollama models.

List Models

Lists all available models.

```
Ollama := TsgcHTTP_API_Ollama.Create(nil);
Ollama.OllamaOptions.Host := 'http://localhost:11434';
WriteLn(Ollama._GetModels);
```

Get Tags

Lists all locally available model tags.

```
Ollama := TsgcHTTP_API_Ollama.Create(nil);
Ollama.OllamaOptions.Host := 'http://localhost:11434';
WriteLn(Ollama._GetTags);
```

Show Model

Retrieves detailed information about a specific model.

```
Ollama := TsgcHTTP_API_Ollama.Create(nil);
Ollama.OllamaOptions.Host := 'http://localhost:11434';
WriteLn(Ollama._ShowModel('llama3'));
```

Pull Model

Downloads a model from the Ollama model library.

```
Ollama := TsgcHTTP_API_Ollama.Create(nil);
Ollama.OllamaOptions.Host := 'http://localhost:11434';
WriteLn(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.

```
Ollama := TsgcHTTP_API_Ollama.Create(nil);
Ollama.OllamaOptions.Host := 'http://localhost:11434';
WriteLn(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.

```
Grok := TsgcHTTP_API_Grok.Create(nil);
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.

```
Grok := TsgcHTTP_API_Grok.Create(nil);
Grok.GrokOptions.ApiKey := 'API_KEY';
WriteLn(Grok._CreateMessage('grok-3', 'Hello!'));
```

System Prompt Example

Send a message with a system prompt to control Grok's behavior.

```
Grok := TsgcHTTP_API_Grok.Create(nil);
Grok.GrokOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Grok := TsgcHTTP_API_Grok.Create(nil);
Grok.GrokOptions.ApiKey := 'API_KEY';
Grok.OnHTTPAPISSE := OnSSEEvent;
Grok._CreateMessageStream('grok-3', 'Tell me a story.');

procedure TForm1.OnSSEEvent(Sender: TObject; const aEvent, aData: string;
  var Cancel: Boolean);
begin
  Memo1.Lines.Add(aData);
end;
```

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.

```
Grok := TsgcHTTP_API_Grok.Create(nil);
Grok.GrokOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Grok := TsgcHTTP_API_Grok.Create(nil);
Grok.GrokOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Mistral := TsgcHTTP_API_Mistral.Create(nil);
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.

```
Mistral := TsgcHTTP_API_Mistral.Create(nil);
Mistral.MistralOptions.ApiKey := 'API_KEY';
WriteLn(Mistral._CreateMessage('mistral-large-latest', 'Hello!'));
```

System Prompt Example

Send a message with a system prompt to control Mistral's behavior.

```
Mistral := TsgcHTTP_API_Mistral.Create(nil);
Mistral.MistralOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Mistral := TsgcHTTP_API_Mistral.Create(nil);
Mistral.MistralOptions.ApiKey := 'API_KEY';
Mistral.OnHTTPAPISSE := OnSSEEvent;
Mistral._CreateMessageStream('mistral-large-latest', 'Tell me a story.');

procedure TForm1.OnSSEEvent(Sender: TObject; const aEvent, aData: string;
  var Cancel: Boolean);
begin
  Memo1.Lines.Add(aData);
end;
```

JSON Mode Example

Request a JSON-formatted response from Mistral.

```
Mistral := TsgcHTTP_API_Mistral.Create(nil);
Mistral.MistralOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Mistral := TsgcHTTP_API_Mistral.Create(nil);
Mistral.MistralOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Mistral := TsgcHTTP_API_Mistral.Create(nil);
Mistral.MistralOptions.ApiKey := 'API_KEY';
WriteLn(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.

```
Mistral := TsgcHTTP_API_Mistral.Create(nil);
Mistral.MistralOptions.ApiKey := 'API_KEY';
WriteLn(Mistral._CreateEmbeddings('mistral-embed', 'Hello world'));
```

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

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.
- **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):

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.

**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 := TsgcIoTAmazon_MQTT_Client.Create(nil);
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;

procedure OnMQTTConnect(Connection: TsgcWSConnection; const Session: Boolean; const ReturnCode: TmqttConnReturnCc
```

```
begin
  ShowMessage('Connected to AWS');
end;
```

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 := TsgcIoTAmazon_MQTT_Client.Create(nil);
...
oClient.OnSubscribe := OnSubscribeEvent;

vPacketIdentifier := oClient.Subscribe('Sensor/moisture/room1');

procedure OnMQTTSubscribe(Connection: TsgcWSConnection; aPacketIdentifier: Word; aCodes: TsgcWSSUBACKS);
begin
  if vPacketIdentifier = aPacketIdentifier then
    ShowMessage('Subscribed to topic Sensor/moisture/room1');
end;

// 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.
procedure OnMQTTPublish(Connection: TsgcWSConnection; aTopic, aText: string);
begin
  DoLog('Received Message: ' + aTopic + ' ' + aText);
end;
```

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

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:

```
oClient := TsgcIoTAzure_MQTT_Client.Create(nil);
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;

procedure OnMQTTConnect(Connection: TsgcWSConnection; const Session: Boolean; const
ReturnCode: TmqttConnReturnCode);
```

```
begin
  ShowMessage('Connected to Azure IoT Hub');
end;
```

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.

```
oProperties := TStringList.Create;
Try
  oProperties.AddPair('prop_name1', 'prop_value1');
  oProperties.AddPair('prop_name2', 'prop_value2');
  oClient.Send_DeviceToCloud('{"temp": 10}', oProperties, azuIoTQoS1);
Finally
  oProperties.Free;
End;
```

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

```
oProperties := TStringList.Create;
Try
  oProperties.AddPair('$.ct', 'application/json');
  oProperties.AddPair('$.ce', 'utf-8');
  oClient.Send_DeviceToCloud('{"temp": 10}', oProperties, azuIoTQoS1);
Finally
  oProperties.Free;
End;
```

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).

```
procedure TFRMSGCClientIoT.AzureIoTMQTTPublish(Connection: TsgcWSConnection; aTopic, aText: string);
begin
  DoLog('Received Message: ' + aTopic + ' ' + aText);
end;
```

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).

```
procedure UploadFileToAzure;
begin
  oDialog := TOpenDialog.Create(nil);
  Try
    if oDialog.Execute then
      AzureIoT.UploadFile(oDialog.FileName);
  Finally
    oDialog.Free;
  End;
end;
```

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.

```
oClient := TsgcIoTAzure_MQTT_Client.Create(nil);
Try
  oClient.Certificate.CertFile := 'cert.pem';
  oClient.Certificate.KeyFile := 'key.pem';
  oClient.Certificate.Enabled := True;
  oResponse := TsgcIoT_Azure_OperationRegistrationState.Create;
  Try
    if oClient.ProvisioningDeviceClient_Register('scope_id', 'registration_id', oResponse) then
      ShowMessage('#Provisioning Register OK: ' + oResponse.Status)
    else
      ShowMessage('#Provisioning Register Error: ' + oResponse.Status);
  Finally
    FreeAndNil(oResponse);
  End;
Finally
  FreeAndNil(oClient);
End;
```

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.
- **HTTP/1 Client:** a non-visual component that inherits from the TIdHTTP client component.
- **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:

```
oClient := TsgcHTTP2Client.Create(nil);
oClient.OnHTTP2Response := OnHTTP2ResponseEvent;
oClient.Get('https://www.google.com');

procedure OnHTTP2ResponseEvent(Sender: TObject; const
    Connection: TsgcHTTP2ConnectionClient; const Request:
    TsgcHTTP2RequestProperty; const Response: TsgcHTTP2ResponseProperty);
begin
  ShowMessage(Response.DataString);
end;
```

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.

```
oClient := TsgcHTTP2Client.Create(nil);
oClient.OnHTTP2Response := OnHTTP2ResponseEvent;
oClient.GetAsync('https://www.gooogle.com');
procedure OnHTTP2ResponseEvent(Sender: TObject; const Connection: TsgcHTTP2ConnectionClient;
  const Request: TsgcHTTP2RequestProperty; const Response: TsgcHTTP2ResponseProperty);
begin
  ShowMessage(Response.Headers.Text + #13#10 + Response.DataString);
end;
```

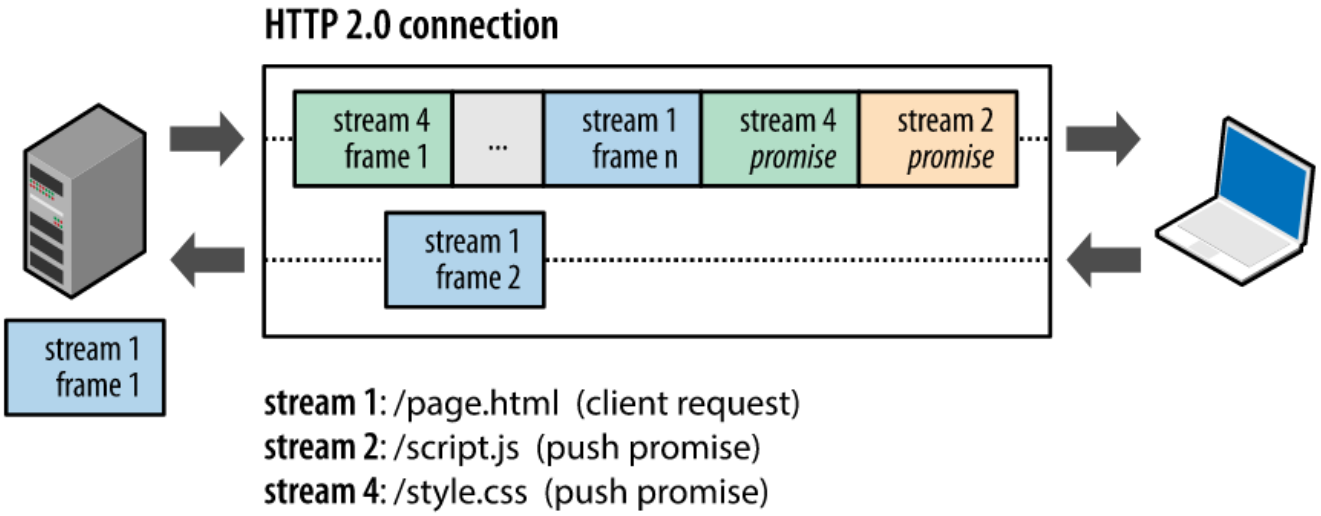
Blocking Mode

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.

```
oClient := TsgcHTTP2Client.Create(nil);
vResponse := oClient.Get('https://www.gooogle.com');
if oClient.Response.Status = 200 then
  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.

```
oClient := TsgcHTTP2Client.Create(nil);
oClient.OnHTTP2PushPromise := OnHTTP2PushPromiseEvent;
oClient.Get('https://http2.golang.org/serverpush');
...
procedure OnHTTP2PushPromiseEvent(Sender: TObject; const Connection: TsgcHTTP2ConnectionClient;
  const PushPromise: TsgcHTTP2_Frame_PushPromise; var Cancel: Boolean);
begin
  if PushPromise.URL = '/serverpush/static/godocs.js' then
    Cancel := True
  else
    Cancel := False;
end;
```

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:

```
oClient := TsgcHTTP2Client.Create(nil);
oClient.HTTP2Options.ReadTimeout := 0; // no timeout for large files
oClient.Get('https://server/largefile', oStream);
```

```
oClient := TsgcHTTP2Client.Create(nil);
oClient.OnHTTP2Response := OnHTTP2ResponseEvent;
oClient.Get('https://http2.golang.org/file/gopher.png');
...
procedure OnHTTP2ResponseEventEvent(Sender: TObject; const Connection: TsgcHTTP2ConnectionClient;
  const Request: TsgcHTTP2RequestProperty; const Response: TsgcHTTP2ResponseProperty)
begin
  oStream := TFileStream.Create('file', fmOpenWrite or fmCreate);
  Try
    oStream.CopyFrom(Response.Data, Response.Data.Size);
  Finally
    oStream.Free;
  End;
end;
```

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.

```
oClient := TsgcHTTP2Client.Create(nil);
oClient.OnHTTP2ResponseFragment := OnHTTP2ResponseFragmentEvent;
oClient.Get('https://http2.golang.org/clockstream');
...
procedure OnHTTP2ResponseFragmentEvent(Sender: TObject; const Connection: TsgcHTTP2ConnectionClient;
  const Request: TsgcHTTP2RequestProperty; const Fragment: TsgcHTTP2ResponseFragmentProperty);
begin
  ShowMessage(Fragment.DataString);
end;
```

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"

```
procedure OnHTTP2BeforeRequest(Sender: TObject; const Connection: TsgcHTTP2ConnectionClient;
  var Headers: TStringList);
begin
  Headers.Add('Client: sgcWebSockets');
end;
```

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 := TsgcHTTP2Client.Create(nil);
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 := TsgcHTTP2Client.Create(nil);
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.

```
oClient := TsgcHTTP2Client.Create(nil);
oClient.OnHTTP2Authorization := OnHTTP2AuthorizationEvent;
...
procedure OnHTTP2AuthorizationEvent(Sender: TObject; const Connection: TsgcHTTP2ConnectionClient; const AuthType,
begin
  if AuthType = 'Basic' then
  begin
    UserName := 'user';
        Password := 'secret';
  end;
end;
```

Bearer Token

If server returns a header requesting Bearer Token Authentication, set OnHTTP2Authorization the token.

```
oClient := TsgcHTTP2Client.Create(nil);
oClient.OnHTTP2Authorization := OnHTTP2AuthorizationEvent;
...
procedure OnHTTP2AuthorizationEvent(Sender: TObject; const Connection: TsgcHTTP2ConnectionClient; const AuthType,
begin
  if AuthType = 'Bearer' then
  begin
    aToken := 'bearer token';
  end;
end;
```

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).

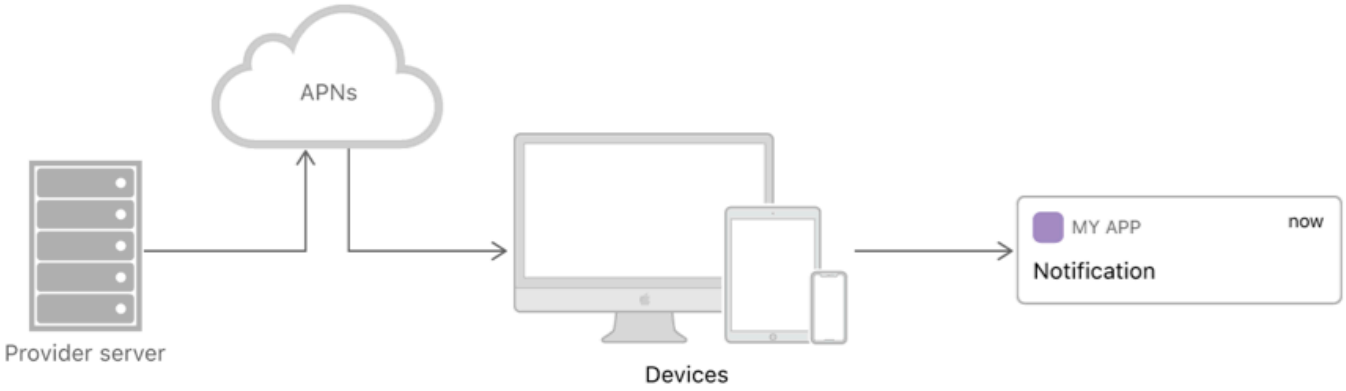
HTTP2 | Apple Push Notifications

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 | Register your APP with APNs

https://developer.apple.com/documentation/usernotifications/registering_your_app_with_apns

You must know which is the device token which identifies the user's device where you want to send the notifications. This Device Token is unique for user and device.

To get the device token using Rad Studio follow the next steps:

1. Make use in your iOS Project of **FMX.PushNotification.iOS** and **System.PushNotification** units.
2. Use **TPushServiceManager** class to register your application and get the device token used.

```
var
  oPushService: TPushService;
  oPushConnection: TPushServiceConnection;
begin
  oPushService := TPushServiceManager.Instance.GetServiceByName(TPushService.TServiceNames.APS
  oPushConnection := TPushServiceConnection.Create(oPushService);
  oPushConnection.Active := True;
  oPushConnection.OnChange := OnChangeEvent;
  oPushConnection.OnReceiveNotification := OnReceiveNotificationEvent;
  vDeviceId := oPushService.DeviceIDValue[TPushService.TDeviceIDNames.DeviceID];
  vDeviceToken := oPushService.DeviceTokenValue[TPushService.TDeviceTokenNames.DeviceToken];
end;



procedure OnChangeEvent(Sender: TObject; AChange: TPushService.TChanges);
begin
  memoLog.Lines.Add('OnChange');
end;



procedure OnReceiveNotificationEvent(Sender: TObject; const ANotification: TPushServiceNotifica
begin
  memoLog.Lines.Add('DataKey=' + ANotification.DataKey);
  memoLog.Lines.Add('JSON=' + ANotification.JSON.ToString);
  memoLog.Lines.Add('DataObject=' + ANotification.DataObject.ToString);
end;
```

The prior code basically creates a new connection with Apple Servers, when connection is fully established, returns a **device token** that is the value used by sgcWebSockets service provider to send notifications to this user's device.

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.

```
oHTTP := TsgcHTTP2Client.Create(nil);
Try
  // ... requires authorization code
  oStream := TStringStream.Create('{"aps":{"alert":"Alert from sgcWebSockets!"}}',
    TEncoding.UTF8);
  Try
    oHTTP.Post('https://api.push.apple/3/device/device_token', oStream);
    if oHTTP.Response.Status = 200 then
      ShowMessage('Notification Sent Successfully')
    else
      ShowMessage('Notification error');
  Finally
    oStream.Free;
  End;
Finally
  oHTTP.Free;
End;
```

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).

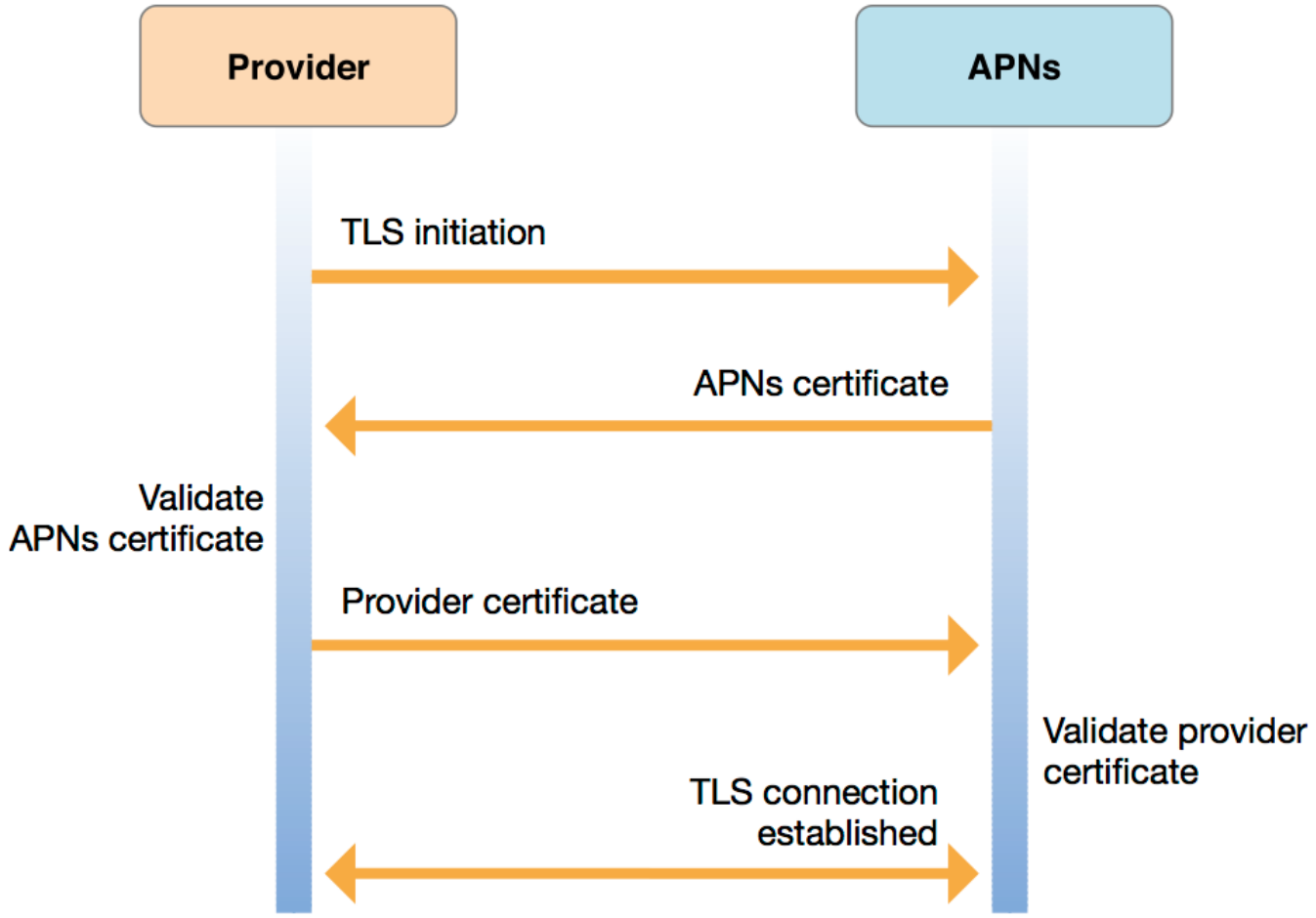
Using Token-Based connections requires sending the **apns-topic** with the value of your app's bundle ID/app id (example: com.example.application).

```
oHTTP := TsgcHTTP2Client.Create(nil);
oHTTP.TLSOptions.IOHandler := iohOpenSSL;



oJWT := TsgcHTTP_JWT_Client.Create(nil);
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.

```
oHTTP := TsgcHTTP2Client.Create(nil);
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.

```
oHTTP := TsgcHTTP2Client.Create(nil);
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');
```

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: by default uses TLS 1.0. If the server requires a higher TLS version, it can be selected here.
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.

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.
- **MinVersion:** 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.

```
SSLOptions.Version := tls1_3;
SSLOptions.OpenSSL_Options.MinVersion := 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 power-shell.
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**.

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**.

```
procedure OnAsyncResultEvent(Sender: TObject; const aRequest:
    TsgcHTTPAsyncRequest; const aResponse: TIdHTTPResponse);
```

If there is any error while processing the Asynchronous request, the exception will be raised in the event **OnAsyncException**.

Examples

Request a **GET** method to HTTPs server and using **TLS 1.2**

```
oHTTP := TsgcHTTP1Client.Create(nil);
Try
  oHTTP.TLSOptions.Version := tls1_2;
  ShowMessage(oHTTP.Get('https://www.google.es'));
Finally
  oHTTP.Free;
End;
```

Request a **GET** method to HTTPs server using **openSSL 1.1** and **TLS 1.3**

```
oHTTP := TsgcHTTP1Client.Create(nil);
Try
  oHTTP.TLSOptions.OpenSSL_Options.APIVersion := oslAPI_1_1;
  oHTTP.TLSOptions.Version := tls1_3;
  ShowMessage(oHTTP.Get('https://www.google.es'));
Finally
  oHTTP.Free;
End;
```

Request an **Asynchronous POST** method and read the response using the **OnAsyncResultEvent**.

```
procedure OnAsyncExceptionEvent(Sender: TObject; const aThread:
    TsgcThread; const E: Exception);
begin
  Log(E.Message);
end;
procedure OnAsyncResultEvent(Sender: TObject; const aRequest:
    TsgcHTTPAsyncRequest; const aResponse: TIdHTTPResponse);
begin
  if aResponse.ResponseCode = 200 then
    Log('ok', aRequest.Response)
  else
    Log('error', aRequest.Response);
end;
oHTTP := TsgcHTTP1Client.Create(nil);
oHTTP.OnAsyncResult := OnAsyncResultEvent;
oHTTP.OnAsyncException := OnAsyncResultEvent;
oRequest := TStringStream.Create('body');
oResponse := TStringStream.Create('');
oHTTP.PostAsync('https://localhost/test', oRequest, oResponse);
```

Request a GET method to HTTPs server using **SChannel for Windows**.

```
oHTTP := TsgcHTTP1Client.Create(nil);
Try
  oHTTP.TLSOptions.IOHandler := iohSChannel;
  oHTTP.TLSOptions.Version := tls1_2;
  ShowMessage(oHTTP.Get('https://www.google.es'));
Finally
  oHTTP.Free;
End;
```

Request **SSE** method to get data events

```
oHTTP := TsgcHTTP1Client.Create(nil);
oHTTP.OnSSEMessage := OnSSEMessageEvent;
oHTTP.GetSSE('https://www.yoursite.com/sse');

procedure OnSSEMessageEvent(Sender: TObject; const aMessage: string; var Cancel: Boolean);
begin
  ShowMessage(aMessage);
end;
```

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

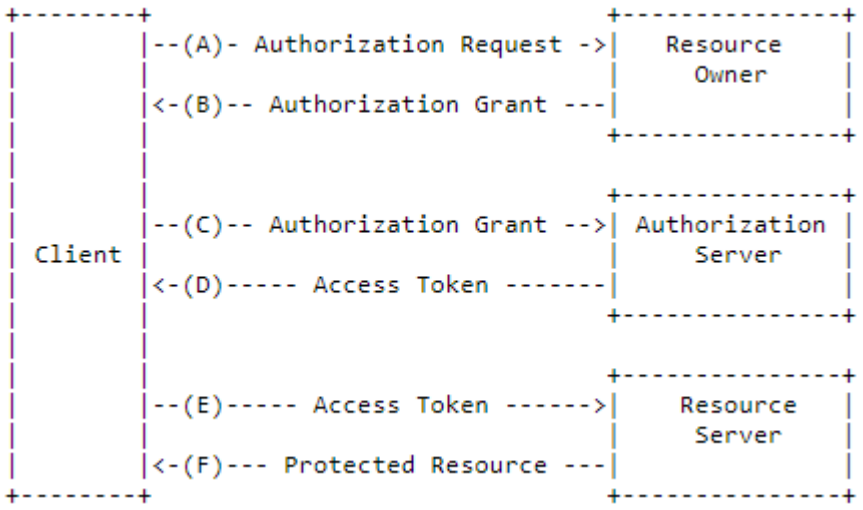
If there is any error while processing an async request, this event is called with the exception raised.

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.

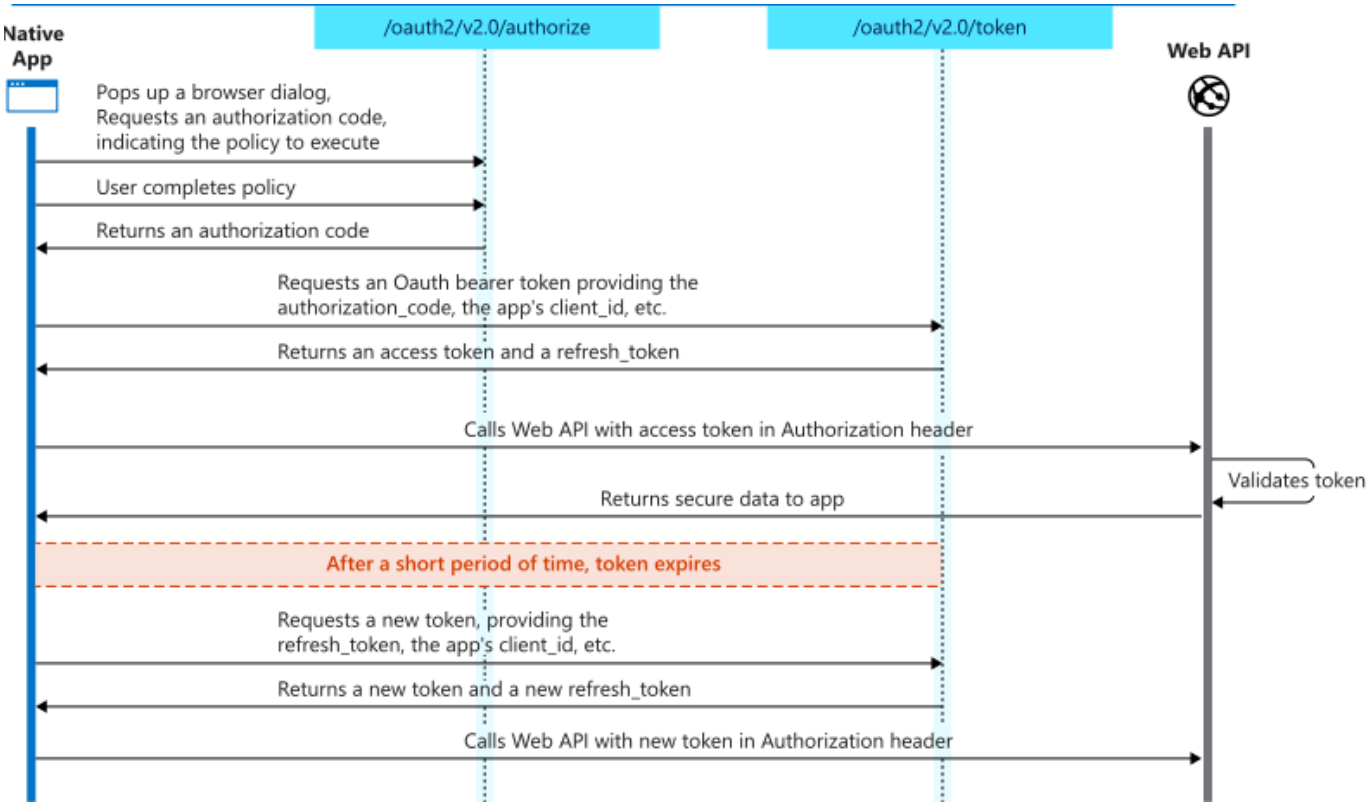
Introduction

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

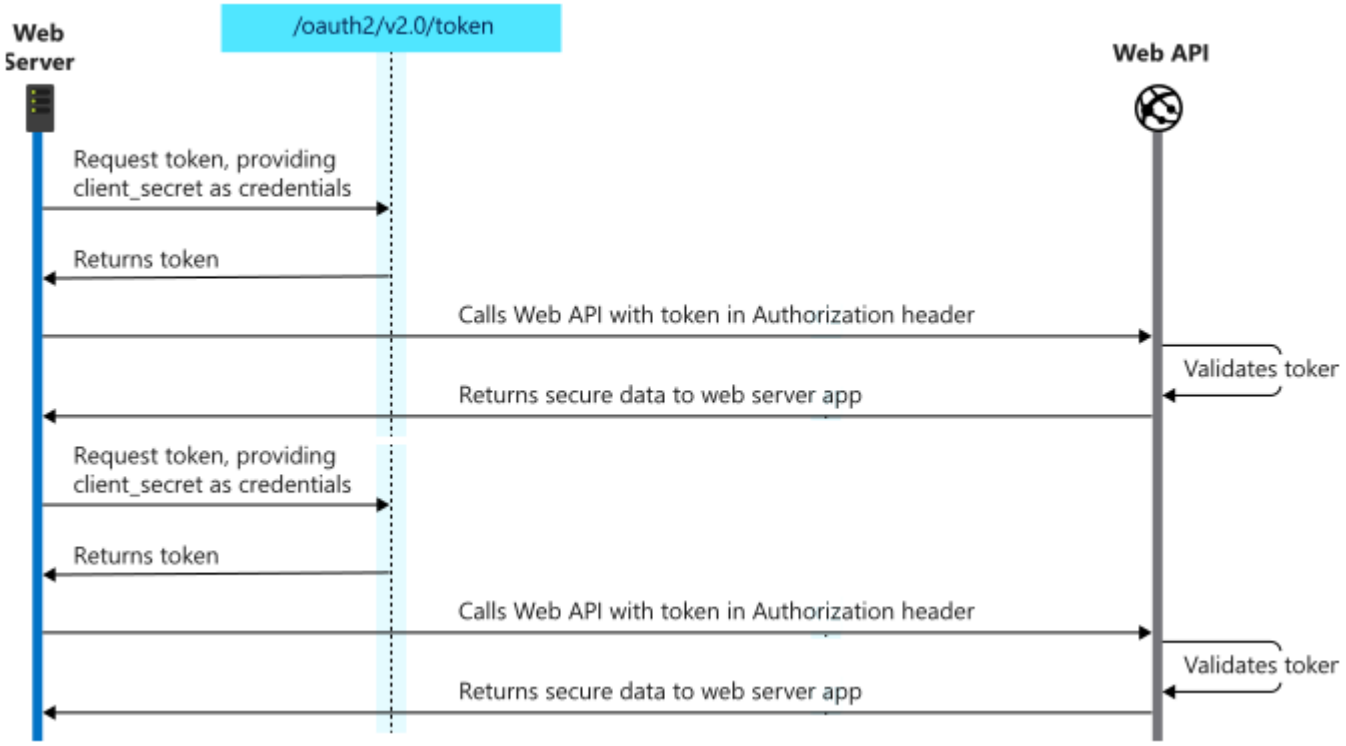
Client supports the following types of Authorization:

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 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

```
procedure GoogleSignIn;
begin
var
  oClient: TsgcHTTP_OAuth2_Client_Google;
  oData: TsgcOAuth2_Google_Data;
begin
  oClient := TsgcHTTP_OAuth2_Client_Google.Create(nil);
  Try
    oData := oClient.Authenticate('client_id', 'client_secret');
    if oData.Authenticated then
      ShowMessage(oData.UserProfile._Name);
  Finally
    oClient.Free;
  End;
end;
```

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

```
procedure MicrosoftSignIn;
begin
var
  oClient: TsgcHTTPComponentClient_OAuth2_Microsoft;
  oData: TsgcOAuth2_Microsoft_Data;
begin
  oClient := TsgcHTTPComponentClient_OAuth2_Microsoft.Create(nil);
  Try
    oData := oClient.Authenticate('tenant_id', 'client_id', 'client_secret');
    if oData.Authenticated then
      ShowMessage(oData.UserProfile.DisplayName);
  Finally
    oClient.Free;
  End;
end;
```

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:

```
OAuth2.Refresh('your-refresh-token');
```

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:

```
OAuth2.Refresh('your-refresh-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:

```
OAuth2.Refresh('your-refresh-token');
```

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

```
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;
```

Handling the Device Code Event

```
procedure TForm1.OAuth2DeviceCode(Sender: TObject;
  const UserCode, VerificationUri: String);
begin
  ShowMessage('Go to ' + VerificationUri + ' and enter code: ' + UserCode);
end;
procedure TForm1.OAuth2DeviceCodeExpired(Sender: TObject);
begin
  ShowMessage('Device code expired. Please try again.');
end;
```

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 (PKCS#11, Windows CNG, etc.)
vCertPEM := MySmartCard.ExportCertificatePEM;

// Convert the certificate to JWK
OAuth2.DPoPOptions.PublicKeyJWK :=
  OAuth2.PublicKeyToJWK(vCertPEM);

// 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:

```
# Generate EC private key (P-256 curve)
openssl ecparam -genkey -name prime256v1 -noout -out dpop_private.pem
# Extract public key
openssl ec -in dpop_private.pem -pubout -out dpop_public.pem
```

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;
// PublicKeyJWK must still be set - read it from the card's certificate
OAuth2.DPoPOptions.PublicKeyJWK :=
  '{"kty":"EC","crv":"P-256","x":"...","y":"..."}';

// Handle signing via smart card
OAuth2.OnDPoPSign := OnDPoPSignHandler;
OAuth2.Start;

// ...

procedure TForm1.OnDPoPSignHandler(Sender: TObject;
```

```
  const SigningInput, Algorithm: String;
  var Signature: String; var Handled: Boolean);
var
  vHash, vSigBytes: TBytes;
begin
  // 1. Hash the signing input (ES256 uses SHA-256)
  vHash := GetHashSHA256(SigningInput);

  // 2. Send the hash to the smart card for signing
  //    (use your PKCS#11, Windows CNG, or card-specific API)
  vSigBytes := MySmartCardAPI.SignHash(vHash);

  // 3. Return the Base64URL-encoded signature
  Signature := EncodeBase64URL(vSigBytes);
  Handled := True;
end;
```

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 (DPoP works with any 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;
// After obtaining the token, generate DPoP proofs for API calls
var vProof := OAuth2.GetDPoPProof('GET', 'https://api.provider.com/resource');
// Use the proof in your HTTP request:
// Authorization: DPoP <access_token>
// DPoP: <vProof>
```

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.

```
OAuth2 := TsgcHTTP_OAuth2_Server.Create(nil);
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 := TsgcWebSocketHTTPServer.Create(nil);
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:

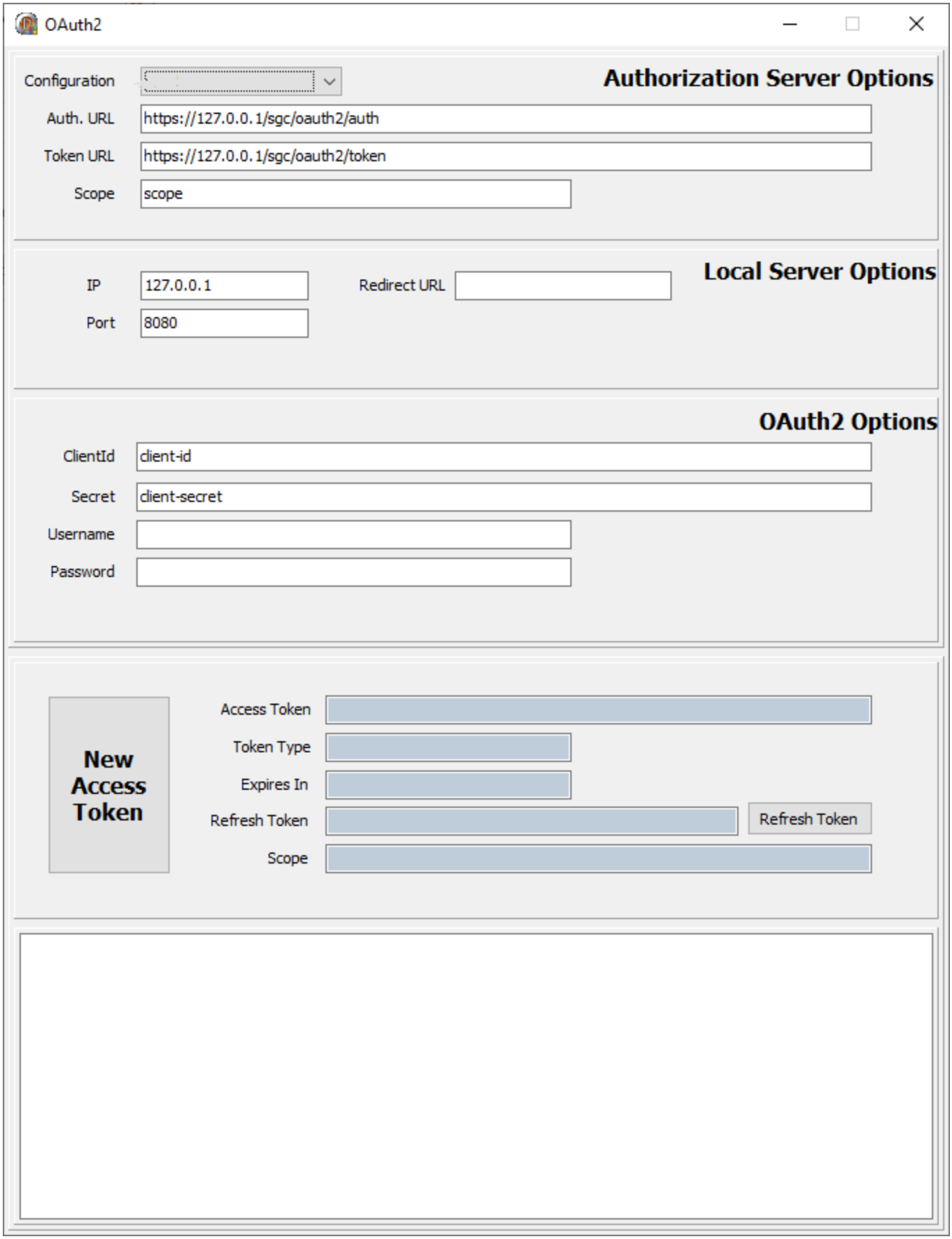
Name: MyApp
RedirectURI: http://127.0.0.1:8080
ClientId: client-id
ClientSecret: client-secret

```
OAuth2 := TsgcHTTP_OAuth2_Server.Create(nil);
OAuth2.Apps.AddApp('MyApp', 'http://127.0.0.1:8080', 'client-id', 'client-secret');
oServer.Authentication.Enabled := True;
oServer.Authentication.OAuth.OAuth2 := OAuth2;
```

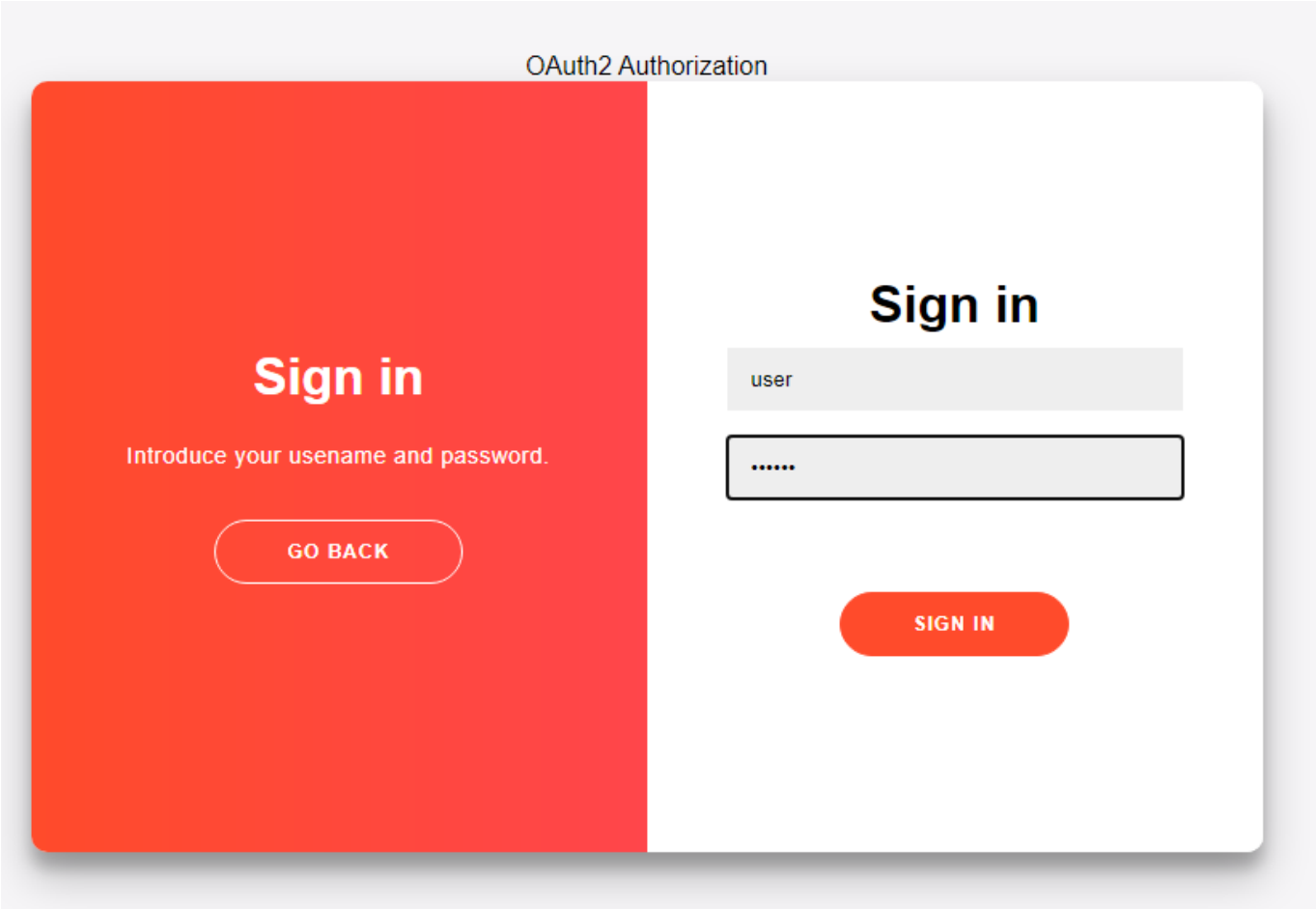
Then handle OnOAuth2Authentication event of OAuth2 server component and implement your own method to login users. I will use the pair "user/secret" to accept a login.

```
procedure OnAuth2Authentication(Connection: TsgcWSConnection; OAuth2: TsgcHTTPOAuth2Request; aUser,
  aPassword: string; var Authenticated: Boolean);
begin
  if ((aUser = 'user') and (aPassword = 'secret')) then
    Authenticated := True;
end;
```

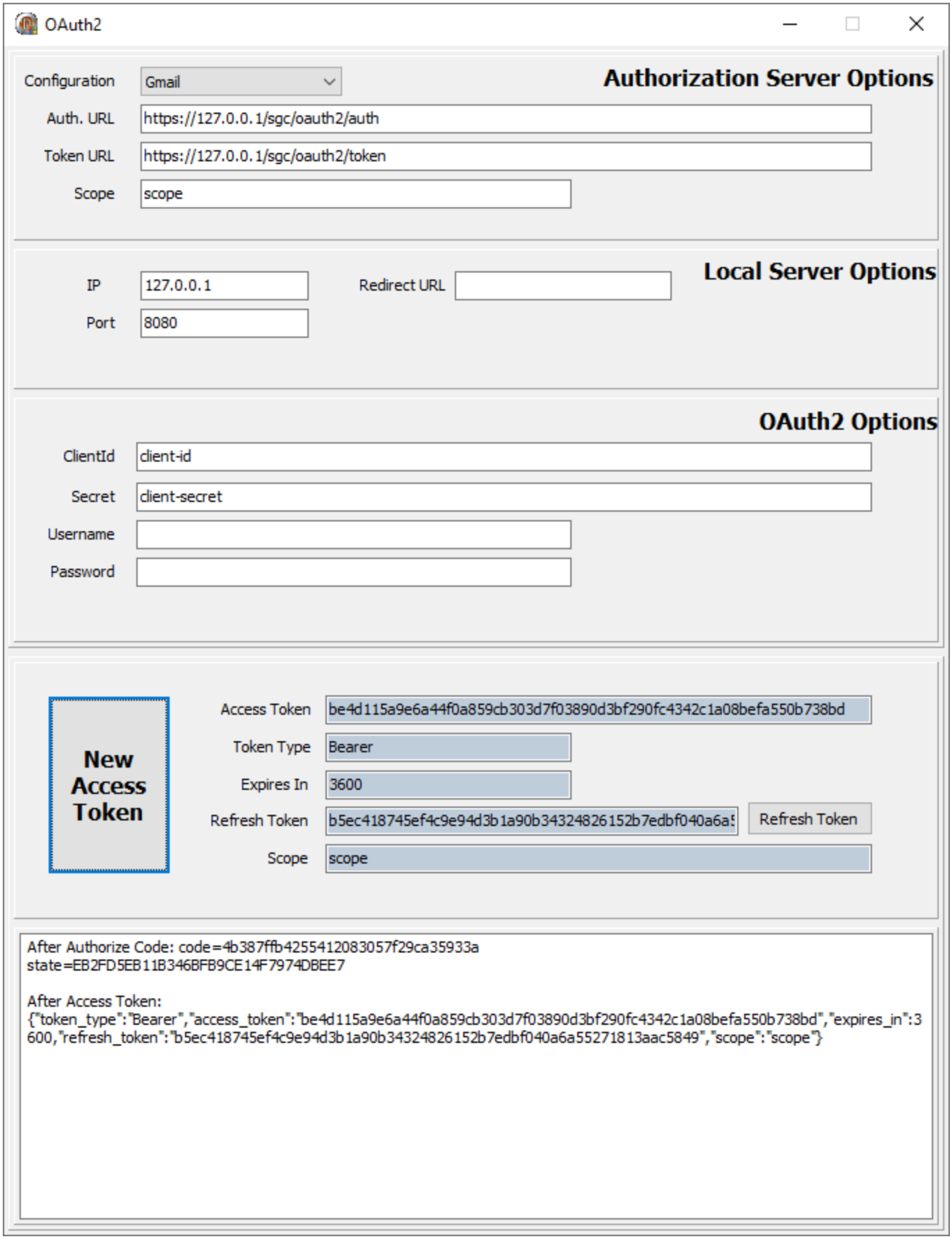
Finally start the server and use a OAuth2 client to login, example you can use the TsgcHTTP_OAuth2_Client included with sgcWebSockets library.



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.

```
procedure OnOAuth2BeforeDispatchPage(Sender: TObject; OAuth2: TsgcHTTPOAuth2Request; var HTML: string);
begin
  HTML := 'your custom html';
end;
```

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:

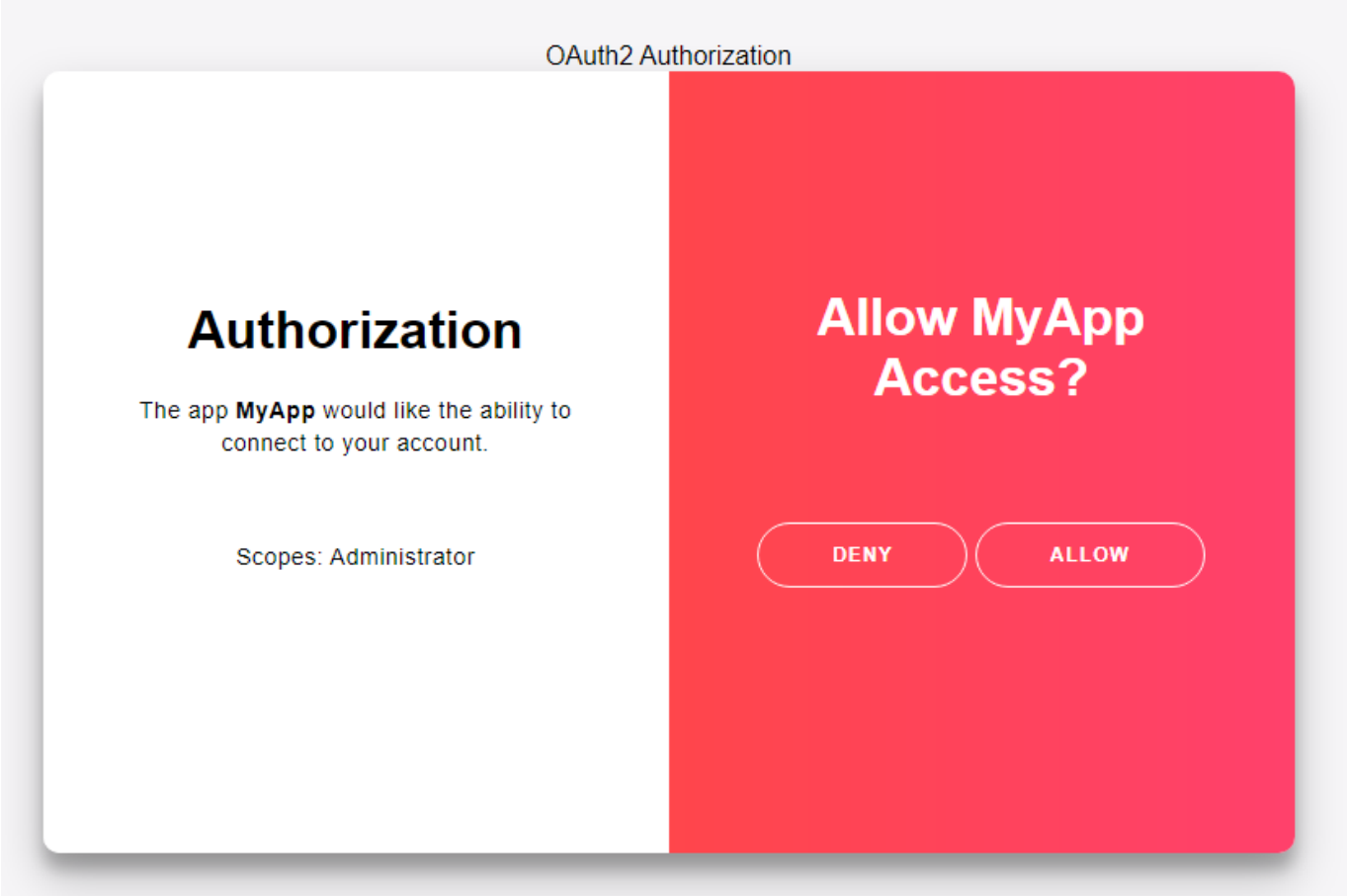
- **AppName:** the name of the app.
- **Token:** access token.
- **Expires:** when the token expires.
- **RefreshToken:** refresh token.

You can save the issued tokens handling the **OAuth2AfterAccessToken** event.

```
procedure OnOAuth2AfterAccessToken(Sender: TObject; Connection: TsgcWSConnection; OAuth2: TsgcHTTPOAuth2Request;
  aResponse: string);
begin
  // ... store OAuth2 Token data
end;

OAuth2 := TsgcHTTP_OAuth2_Server.Create(nil);
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.

```
procedure OnAuth2Authentication(Connection: TsgcWSConnection; OAuth2: TsgcHTTPOAuth2Request; aUser,
  aPassword: string; var Authenticated: Boolean);
begin
  if ((aUser = 'user') and (aPassword = 'secret')) then
    Authenticated := True;
end;
```

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(Sender: TObject; aConnection: TsgcWSConnection; aHeaders: TStringList;
  var Cancel: Boolean);
begin
  if DecodeGETFullPath(aHeaders) = '/Public.html' then
    Cancel := True
  else if DecodePOSTFullPath(aHeaders) = '/Form.html' then
    Cancel := True;
end;
```

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, [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
OAuth2Server := TsgcHTTP_OAuth2_Server.Create(nil);

// Register an application
OAuth2Server.Apps.AddApp('MyApp', 'http://127.0.0.1:8080',
  'my-client-id', 'my-client-secret', 3600, True, [auth2Code]);

// Attach to HTTP server
Server.Authentication.Enabled := True;
Server.Authentication.OAuth.OAuth2 := OAuth2Server;

// Handle authentication
procedure TForm1.OAuth2ServerOAuth2Authentication(Sender: TObject;
  const User, Password: String; var Authenticated: Boolean);
begin
  Authenticated := (User = 'admin') and (Password = 'secret');
end;

// Handle token issuance
procedure TForm1.OAuth2ServerOAuth2AfterAccessToken(Sender: TObject;
  const Token, RefreshToken: String; ExpiresIn: Integer);
begin
  Log('Access token issued: ' + Token);
end;
```

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,
  [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

```
OAuth2Server := TsgcHTTP_OAuth2_Server.Create(nil);
// Register a service application for client credentials
OAuth2Server.Apps.AddApp('BackendService', '',
  'service-client-id', 'service-client-secret', 7200, False,
  [auth2ClientCredentials]);
// Attach to HTTP server
Server.Authentication.Enabled := True;
Server.Authentication.OAuth.OAuth2 := OAuth2Server;
// Log token issuance
procedure TForm1.OAuth2ServerOAuth2AfterAccessToken(Sender: TObject;
  const Token, RefreshToken: String; ExpiresIn: Integer);
begin
  Log('Service token issued: ' + Token);
end;
```

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,
  [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

```
OAuth2Server := TsgcHTTP_OAuth2_Server.Create(nil);
// Register app with password grant
OAuth2Server.Apps.AddApp('MyApp', '',
  'my-client-id', 'my-client-secret', 3600, True,
  [auth2ResourceOwnerPassword]);
// Attach to HTTP server
Server.Authentication.Enabled := True;
Server.Authentication.OAuth.OAuth2 := OAuth2Server;
// Validate user credentials
procedure TForm1.OAuth2ServerOAuth2Authentication(Sender: TObject;
  const User, Password: String; var Authenticated: Boolean);
begin
  // Validate against your user store
  Authenticated := ValidateUser(User, Password);
end;
procedure TForm1.OAuth2ServerOAuth2AfterAccessToken(Sender: TObject;
  const Token, RefreshToken: String; ExpiresIn: Integer);
begin
  Log('Token issued for user: ' + Token);
end;
```

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, [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

```
OAuth2Server := TsgcHTTP_OAuth2_Server.Create(nil);
// Register app with refresh token support
OAuth2Server.Apps.AddApp('MyApp', 'http://127.0.0.1:8080',
  'my-client-id', 'my-client-secret', 3600, True, [auth2Code]);
Server.Authentication.Enabled := True;
Server.Authentication.OAuth.OAuth2 := OAuth2Server;
// Log refresh token usage
procedure TForm1.OAuth2ServerOAuth2AfterRefreshToken(Sender: TObject;
  const OldToken, NewToken, NewRefreshToken: String; ExpiresIn: Integer);
begin
  Log('Token refreshed. New token: ' + NewToken);
end;
```

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

```
OAuth2Server := TsgcHTTP_OAuth2_Server.Create(nil);
// Enable device authorization
OAuth2Server.OAuth2Options.DeviceAuthorization.Enabled := True;
OAuth2Server.OAuth2Options.DeviceAuthorization.ExpiresIn := 600;
OAuth2Server.OAuth2Options.DeviceAuthorization.Interval := 5;
// Register app
OAuth2Server.Apps.AddApp('DeviceApp', '',
  'device-client-id', 'device-client-secret', 3600, True,
  [auth2Code]);
Server.Authentication.Enabled := True;
Server.Authentication.OAuth.OAuth2 := OAuth2Server;
// Handle device code verification
procedure TForm1.OAuth2ServerOAuth2DeviceCodeVerification(Sender: TObject;
  const UserCode: String; var Authorized: Boolean);
begin
  // Validate the user code and authorize the device
  Authorized := True;
end;
procedure TForm1.OAuth2ServerOAuth2AfterAccessToken(Sender: TObject;
  const Token, RefreshToken: String; ExpiresIn: Integer);
begin
  Log('Device authorized. Token: ' + Token);
end;
```

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

```
OAuth2Server := TsgcHTTP_OAuth2_Server.Create(nil);

// Enable token revocation
OAuth2Server.OAuth2Options.Revocation.Enabled := True;

// Register app
OAuth2Server.Apps.AddApp('MyApp', 'http://127.0.0.1:8080',
  'my-client-id', 'my-client-secret', 3600, True, [auth2Code]);

Server.Authentication.Enabled := True;
Server.Authentication.OAuth.OAuth2 := OAuth2Server;

// Handle revocation
procedure TForm1.OAuth2ServerOAuth2AfterRevokeToken(Sender: TObject;
  const Token, TokenTypeHint: String; var Revoked: Boolean);
begin
  Log('Token revoked: ' + Token + ' (type: ' + TokenTypeHint + ')');
  Revoked := True;
end;
```

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

```
OAuth2Server := TsgcHTTP_OAuth2_Server.Create(nil);

// Enable token introspection
OAuth2Server.OAuth2Options.Introspection.Enabled := True;

// Register app
OAuth2Server.Apps.AddApp('MyApp', 'http://127.0.0.1:8080',
  'my-client-id', 'my-client-secret', 3600, True, [auth2Code]);

Server.Authentication.Enabled := True;
Server.Authentication.OAuth.OAuth2 := OAuth2Server;

// Custom introspection validation
procedure TForm1.OAuth2ServerOAuth2AfterIntrospectToken(Sender: TObject;
  const Token: String; var IsActive: Boolean);
begin
  Log('Token introspected: ' + Token + ' Active: ' + BoolToStr(IsActive, True));
  // Optionally override: IsActive := False to force-invalidate
end;
```

Resource Server Request Example

A resource server validates a token by sending a POST request to the introspection endpoint:

```
// Resource server: introspect a token
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.

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

```
OAuth2Server := TsgcHTTP_OAuth2_Server.Create(nil);
// Enable DPoP
OAuth2Server.OAuth2Options.DPoP := True;
// Register app
OAuth2Server.Apps.AddApp('MyApp', 'http://127.0.0.1:8080',
  'my-client-id', 'my-client-secret', 3600, True, [auth2Code]);
Server.Authentication.Enabled := True;
Server.Authentication.OAuth.OAuth2 := OAuth2Server;
// Custom DPoP validation
procedure TForm1.OAuth2ServerOAuth2ValidateDPoP(Sender: TObject;
  const DPoPProof, AccessToken, Method, URL: String; var Valid: Boolean);
begin
  // Additional custom checks
  Log('DPoP proof validated for: ' + Method + ' ' + URL);
  Valid := True; // Set to False to reject the request
end;
```

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.

```
OAuth2Provider := TsgcHTTP_OAuth2_Server_Provider.Create(nil);
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.

```
procedure OnOAuth2IsPrivateEndpoint(Sender: TObject; const aEndpoint: string; var IsPrivate: Boolean);
begin
  if aEndpoint = '/private' then
    IsPrivate := True;
end;
```

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.

```
procedure OnCommandGet(AContext: TIdContext; ARequestInfo: TIdHTTPRequestInfo; AResponseInfo: TIdHTTPResponseInfo
begin
  if ARequestInfo.Document = '/private' then
  begin
    // return OAuth2 profile data
    AResponseInfo.ContentText := OAuth2Provider.Get(ARequestInfo, 'https://graph.microsoft.com/v1.0/me');
    AResponseInfo.ContentType := 'application/json';
    AResponseInfo.ResponseNo := 200;
  end
  else
    AResponseInfo.ResponseNo := 404;
end;
```

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

OpenSSL libraries are required to sign and verify the JWT.

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.
  - **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 you to use 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 you to use 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.

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.

```
oClient := TsgcWebSocketClient.Create(nil);
oClient.URL := 'ws://www.esegece.com:2052';


oJWT := TsgcHTTP_JWT_Client.Create(nil);
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.

```
oHTTP := TsgcHTTP2Client.Create(nil);


oJWT := TsgcHTTP_JWT_Client.Create(nil);
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.

```
oJWT := TsgcHTTP_JWT_Client.Create(nil);
// ... 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.

```
oServer := TsgcWebSocketHTTPServer.Create(nil);
oServer.Port := 80;
oJWT := TsgcHTTP_JWT_Server.Create(nil);
oJWT.JWTOptions.Algorithms.RS.PublicKey.Text := 'public key here';
oServer.Authorization.Enabled := True;
oServer.Authorization.JWT.JWT := oJWT;
oServer.Active := True;
```

Checks property allows you to enable some checks in the Payload of JWT, by default checks if the issued dates are valid.

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

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.

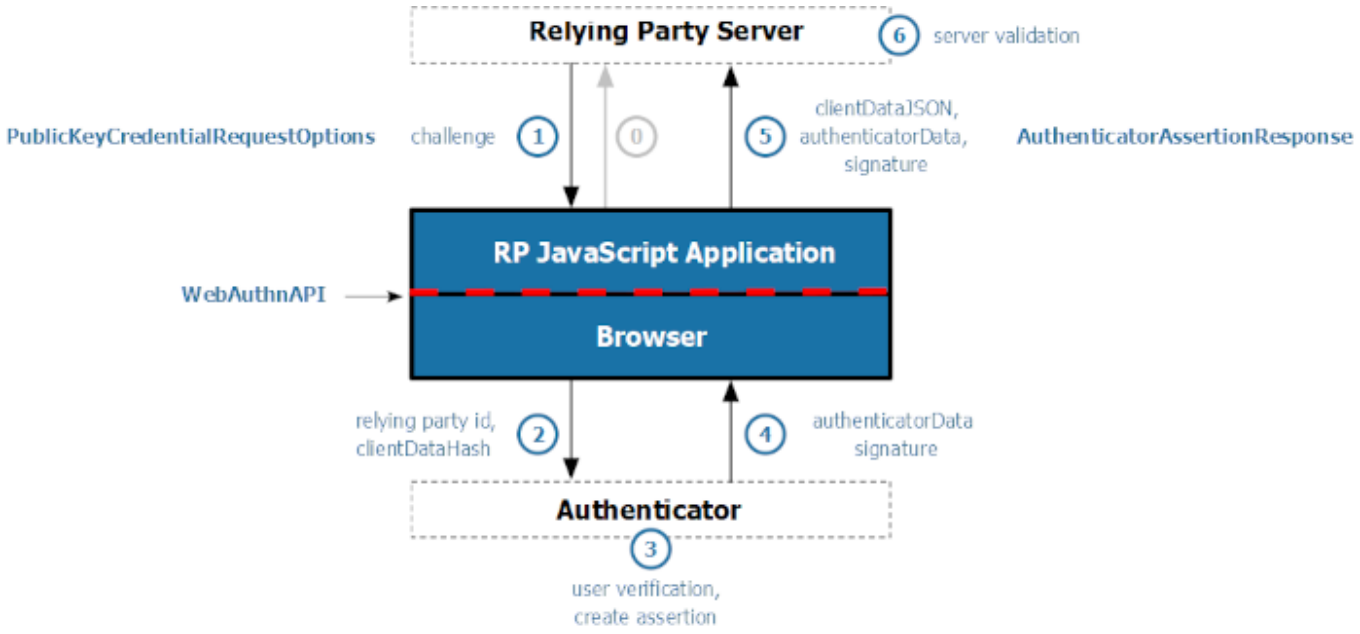
- **Limit:** the max number of credentials that will be sent when ExcludeCredentials is true.

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 — 15 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.

```
procedure OnWebAuthnRegistrationOptionsRequest(Sender: TObject; const aRequest: TsgcWebAuthn_RegistrationOptions_
begin
  if aRequest.Username = 'anonymous' then
    Accept := False;
end;
```

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.

```
procedure OnWebAuthnRegistrationOptionsResponse(Sender: TObject; const aRequest: TsgcWebAuthn_RegistrationOptions
begin
  if aRequest.Username = 'esegece.com' then
  begin
```

```
    aResponse.ExcludeCredentials.AddCredentialRecordFromJSON('json1.txt');
    aResponse.ExcludeCredentials.AddCredentialRecordFromJSON('json2.txt');
  end;
end;
```

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.

```
procedure OnWebAuthnRegistrationSuccessful(Sender: TObject; const aRegistration: TsgcWebAuthn_Registration;
  const aCredentialRecord: TsgcWebAuthn_CredentialRecord; var Accept: Boolean);
begin
  // store in a db
  DB.Credentials.Append;
  DB.Credentials.FieldByName('Credentials').AsString := aCredentialRecord.AsJSON;
  DB.Credentials.Post;
end;
```

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.

```
procedure OnWebAuthnRegistrationError(Sender:
    TObject; const aRequest: TsgcWebAuthn_RegistrationVerify_Request; const
    aRegistration: TsgcWebAuthn_Registration; const aError: string);
begin
  Log('#webauthn_registration_error: ' + aError);
end;
```

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).

```
procedure OnWebAuthnAuthenticationOptionsRequest(
    Sender: TObject; const aRequest:
    TsgcWebAuthn_AuthenticationOptions_Request; var CredentialRecords:
    TsgcWebAuthn_CredentialRecords; var Accept: Boolean);
begin
  if UserExistsInDB(aRequest.Username) then
  begin
    While not EOF do
    begin
      CredentialRecords.AddCredentialRecordFromJSON(RecordFromDB);
      Next;
    end;
  end;
end;
```

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:

```
procedure GetHTTPRequest(const aURL: string; const aToken: string): string;
var
  oHTTP: TsgcHTTP1Client;
begin
  oHTTP := TsgcHTTP1Client.Create(nil);
  Try
    oHTTP.Request.CustomHeaders.AddValue('Authorization', 'Bearer ' + aToken);
    result := oHTTP.Get(aURL);
  Finally
    oHTTP.Free;
  End;
end;
```

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:

```
procedure OnClientHandshake(Connection: TsgcWSConnection; var Headers: TStringList);
begin
        Headers.Add('Authorization: Bearer C760C1C39E3D4E829693A13F18F5CFDE537B516336FC48F7BAB0276176F9E6DE');
end;
```

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!';
}
```

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

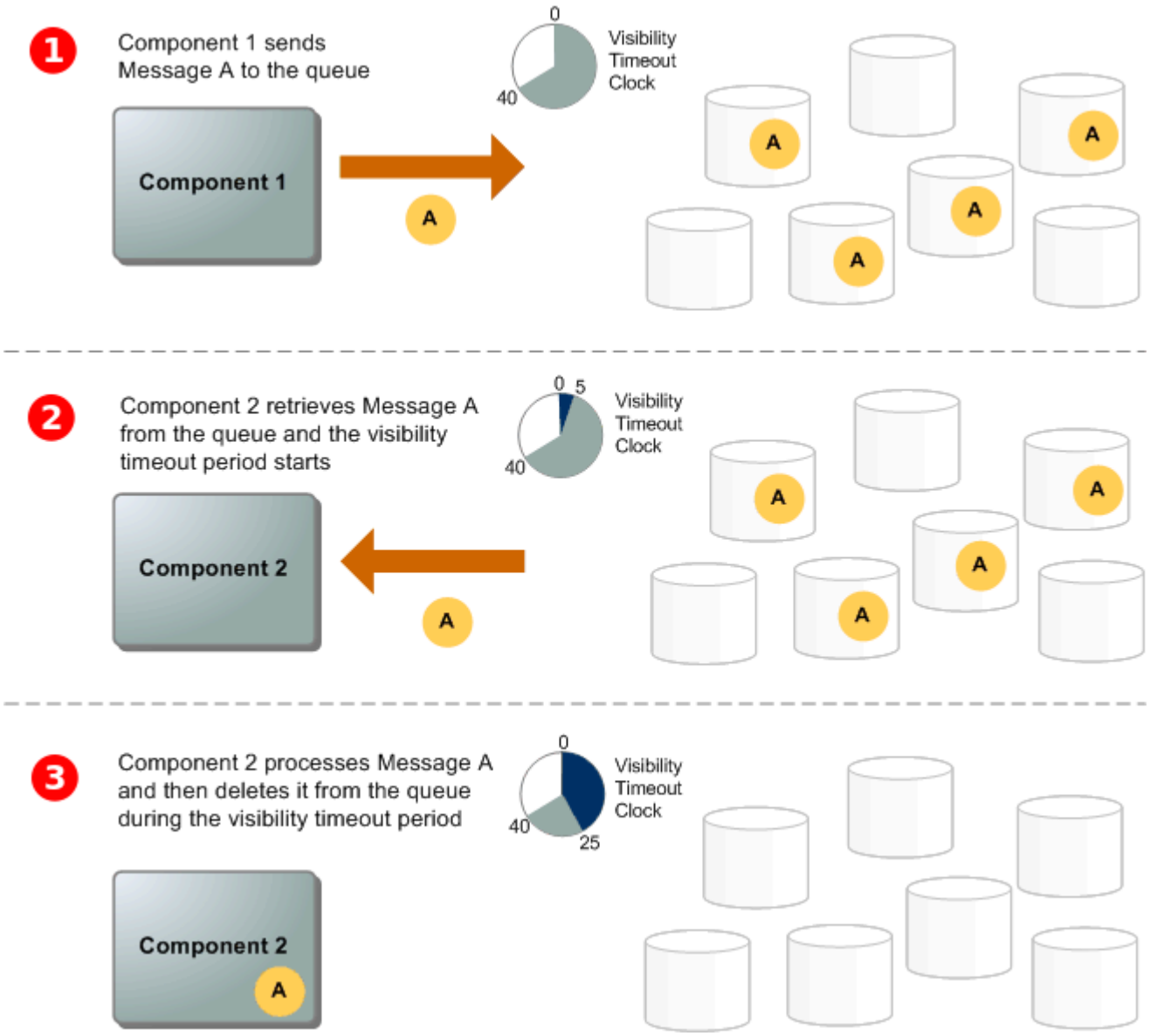
You can use Amazon SQS to exchange sensitive data between applications using server-side encryption (SSE) to encrypt each message body.

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  '' then
    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...') then
    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) then
    DoLog('#Delete Queue: ok')
  else
    DoLog('#Delete Queue: error');

// GetQueueAttributes
// Gets attributes for the specified queue.

  oAttributes := TsgcSQSAttributes.Create;
  Try
    if SQS.GetQueueAttributes('sqs_queue', oAttributes) then
```

```
    begin
      for i := 0 to oAttributes.Count - 1 do
        DoLog('#Attribute: ' + TsgcSQSAttribute(oAttributes->Item[i])
          .AttributeName + ' ' + TsgcSQSAttribute(oAttributes->Item[i])
          .AttributeValue);
    end
    else
      DoLog('#GetQueueAttributes: error');
  Finally
    FreeAndNil(oAttributes);
  End;

// 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') then
    DoLog('#PurgueQueue: ok')
  else
    DoLog('#PurgueQueue: error');

// ReceiveMessage
// Retrieves one or more messages (up to 10), from the specified queue.

  oResponses := TsgcSQSReceiveMessageResponses.Create;
  Try
    if SQS.ReceiveMessage('sqs_test', oResponses) then
    begin
      for i := 0 to oResponses.Count - 1 do
      begin
        DoLog('#ReceiveMessage: ' + TsgcSQSReceiveMessageResponse(oResponses.Item[i]).Body);
        FReceiptHandle := TsgcSQSReceiveMessageResponse(oResponses.Item[i]).ReceiptHandle;
      end;
    end;
  Finally
    FreeAndNil(oResponses);
  End;

// 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') then
    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.

  oAttributes := TsgcSQSAttributes.Create;
  Try
    oAttributes.AddSQSAttribute(sqsatVisibilityTimeout, '45');
    if SQS.SetQueueAttributes('sqs_queue', oAttributes) then
      DoLog('#SetQueueAttributes: ok')
    else
      DoLog('#SetQueueAttributes: error');
  Finally
    FreeAndNil(oAttributes);
  End;

// 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.

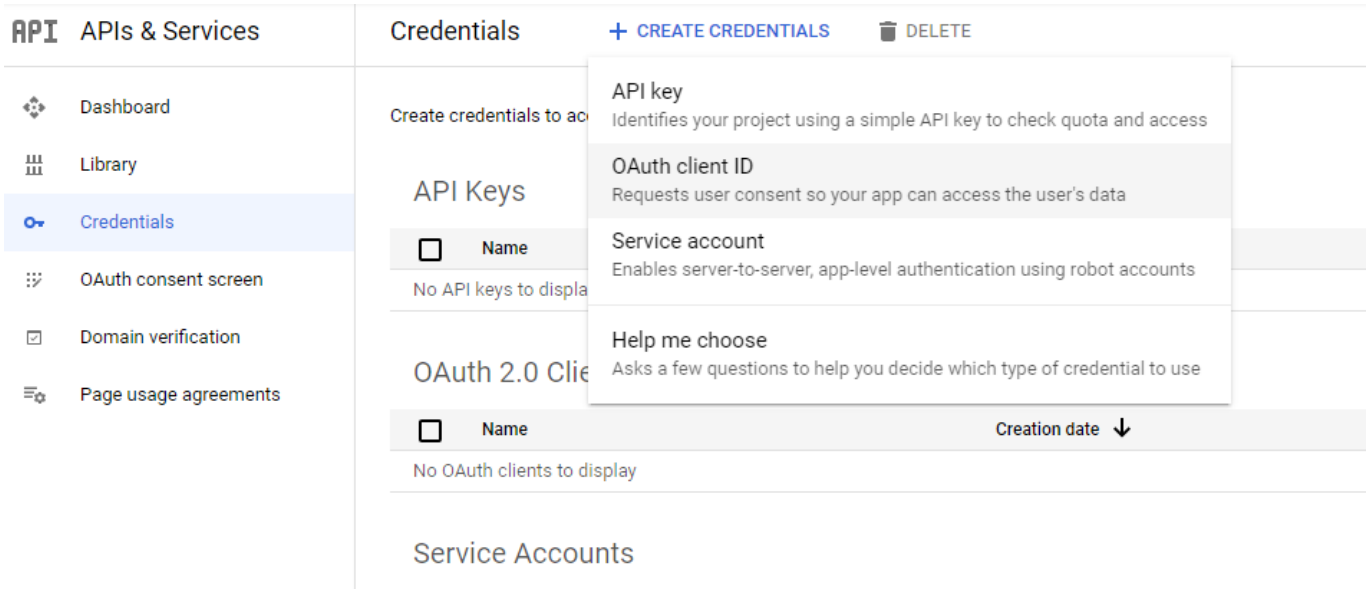
OnSQSResponse

This event is called after an HTTP request with raw response from server.

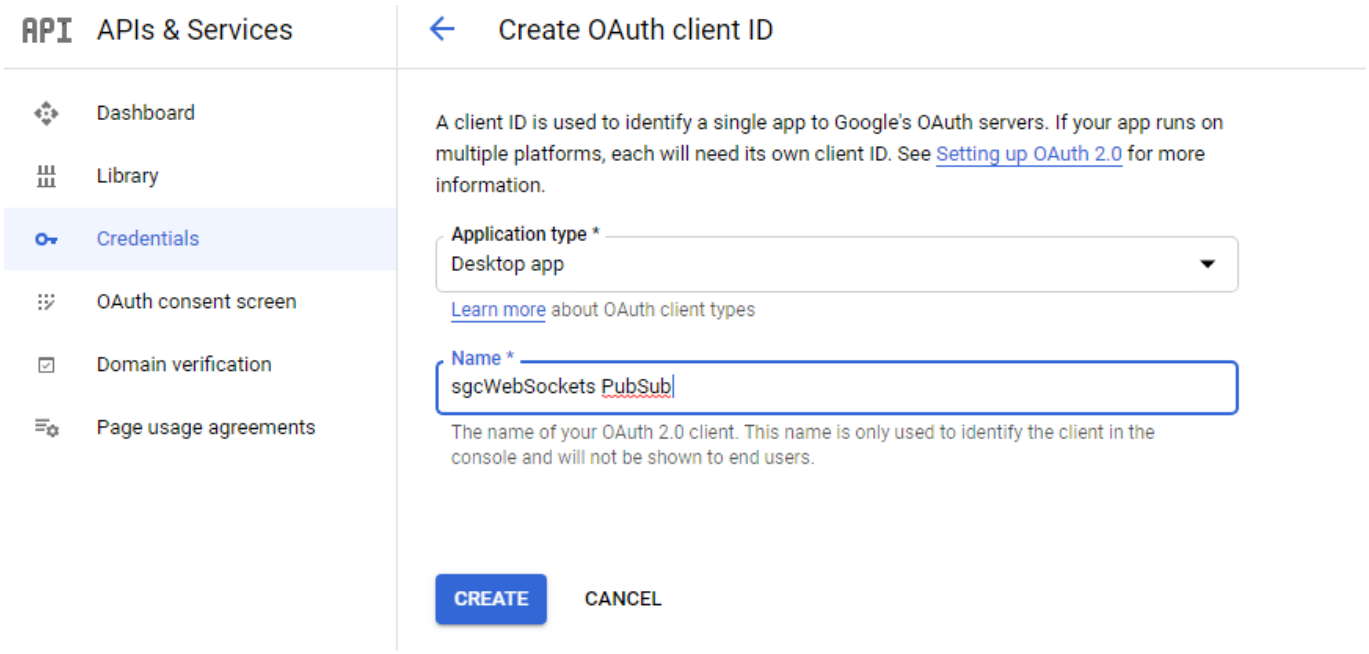
Google Cloud | Google OAuth2 Keys

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.

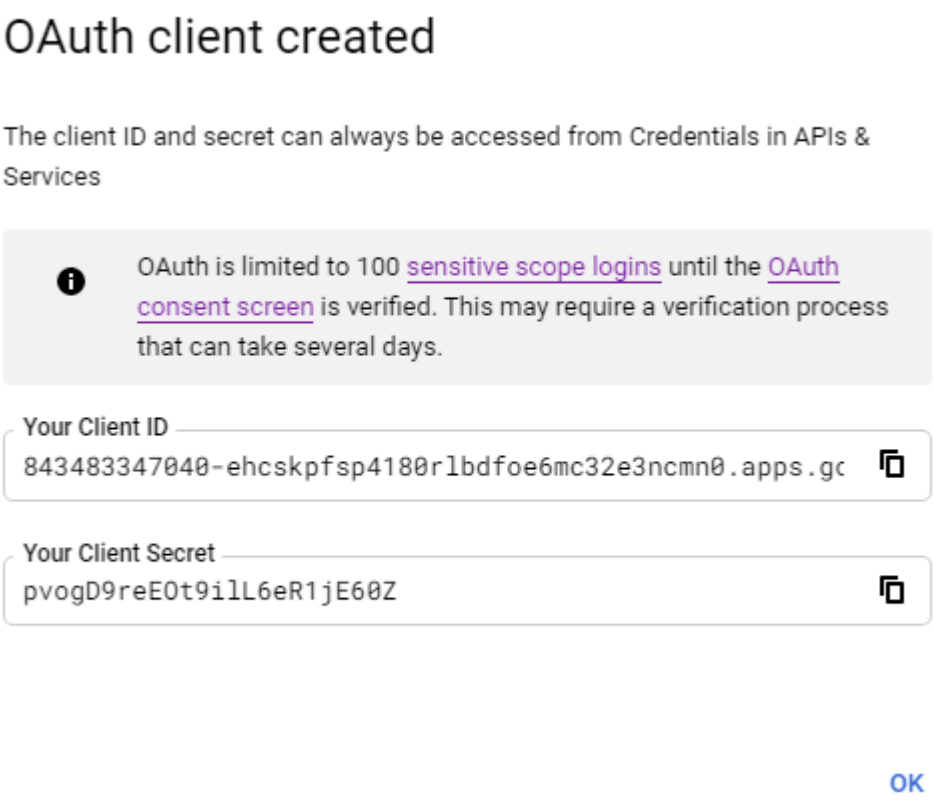
First **login** to your **Google Cloud Account** and use an existing project or create a new one.
After that, go to **Credentials** menu and press the button **CREATE CREDENTIALS**, select the option **OAuth Client ID**.



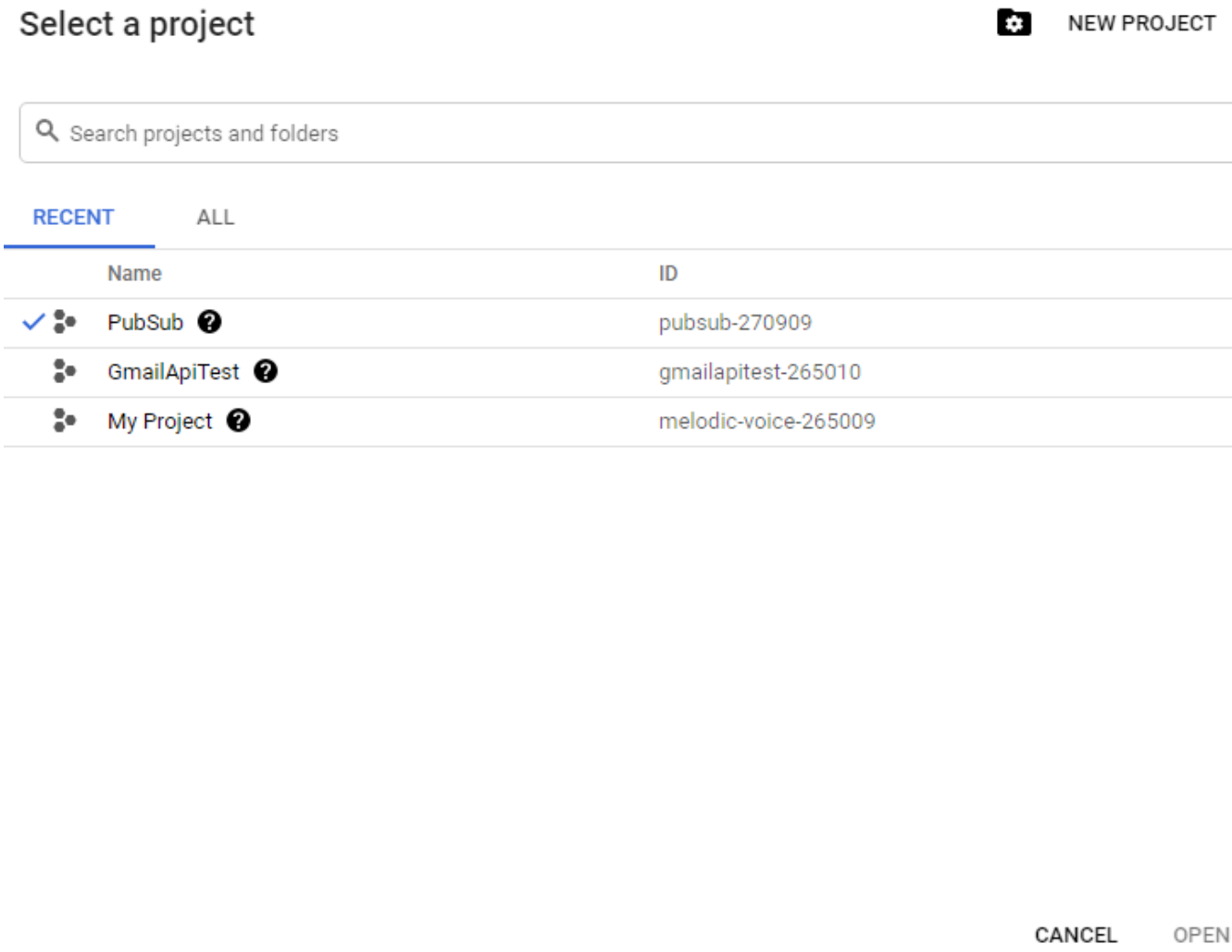
Select your application type and set a descriptive name.



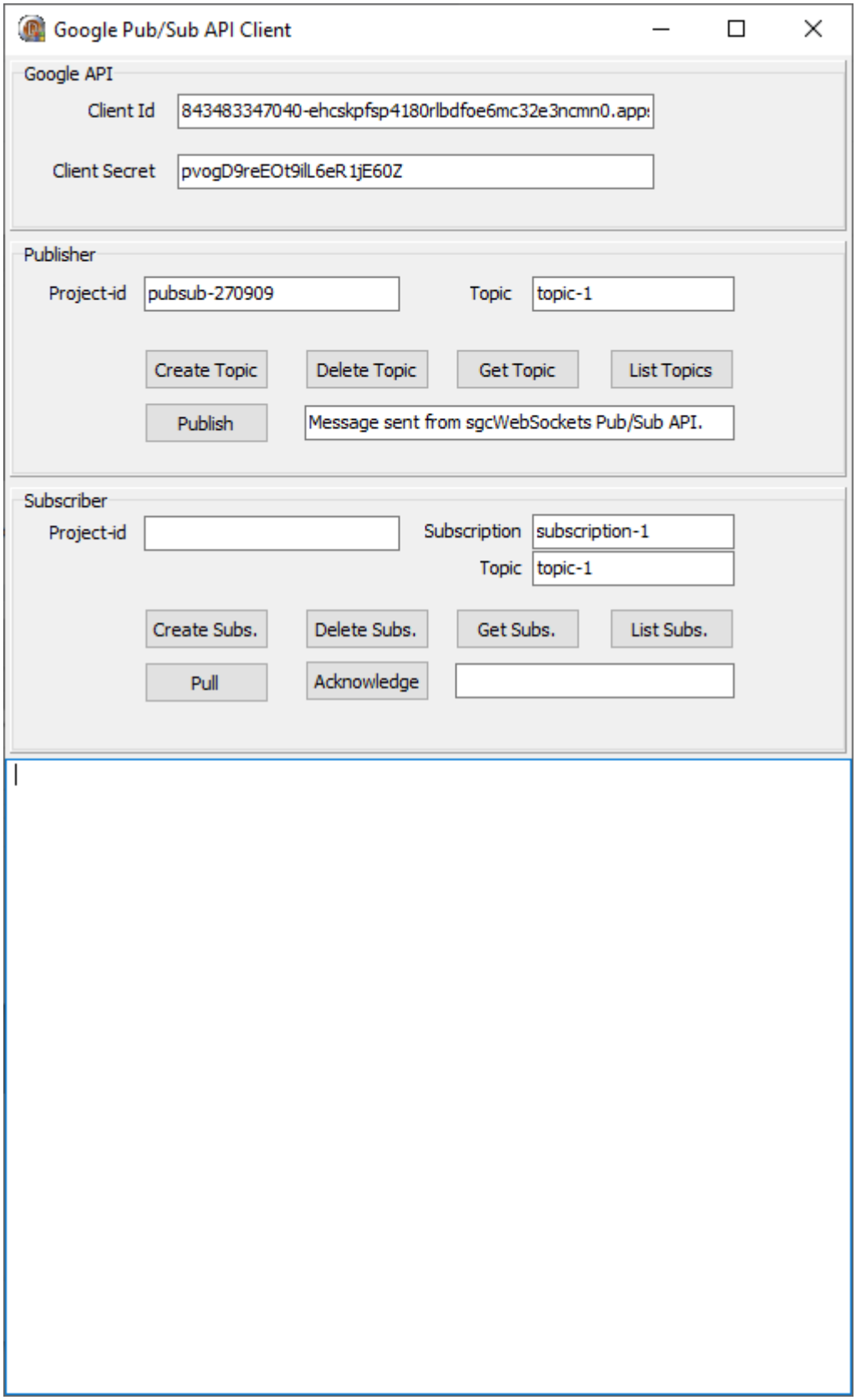
If successful, you will get your Client ID and Client Secret.



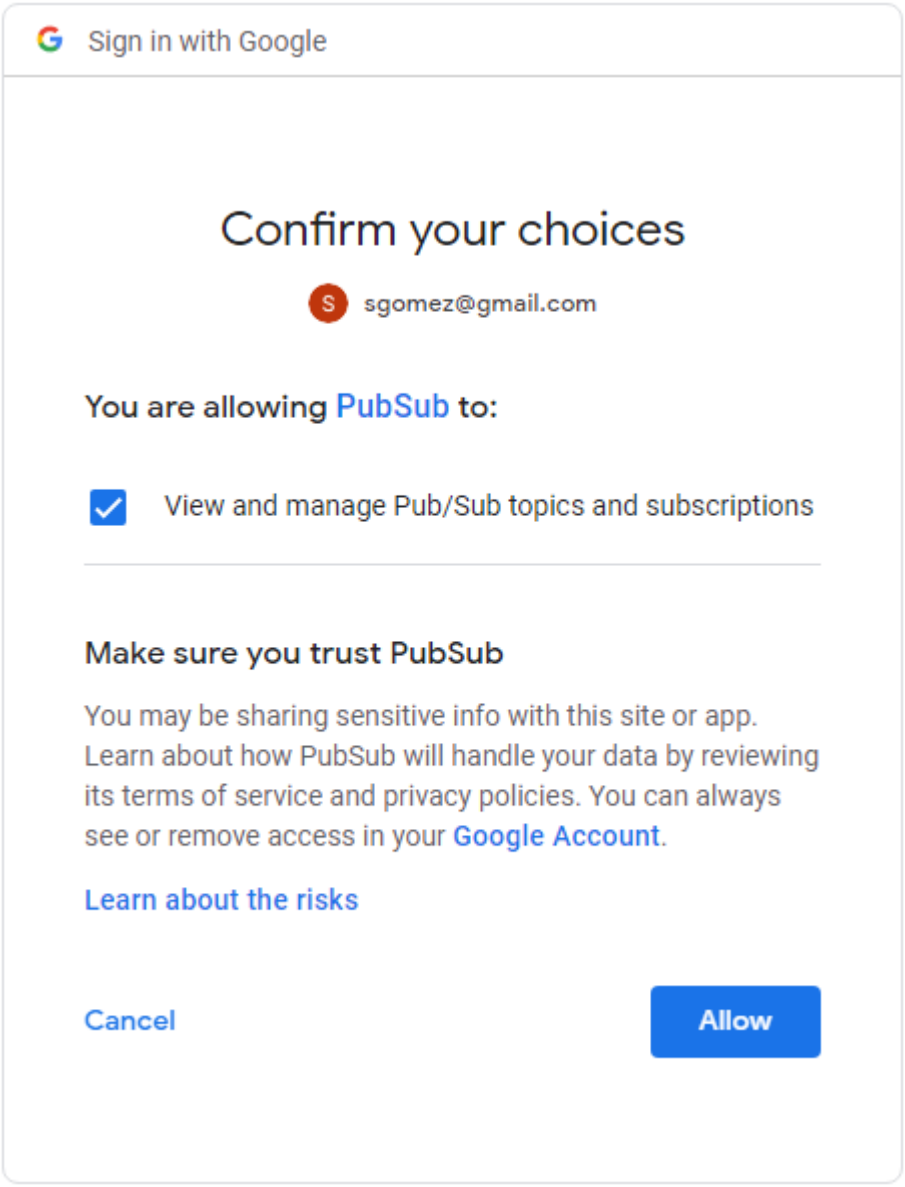
Don't share your OAuth2 data with anyone!
Now copy to the sgcWebSockets PubSub sample, and add the Project Id (NOT the project name)



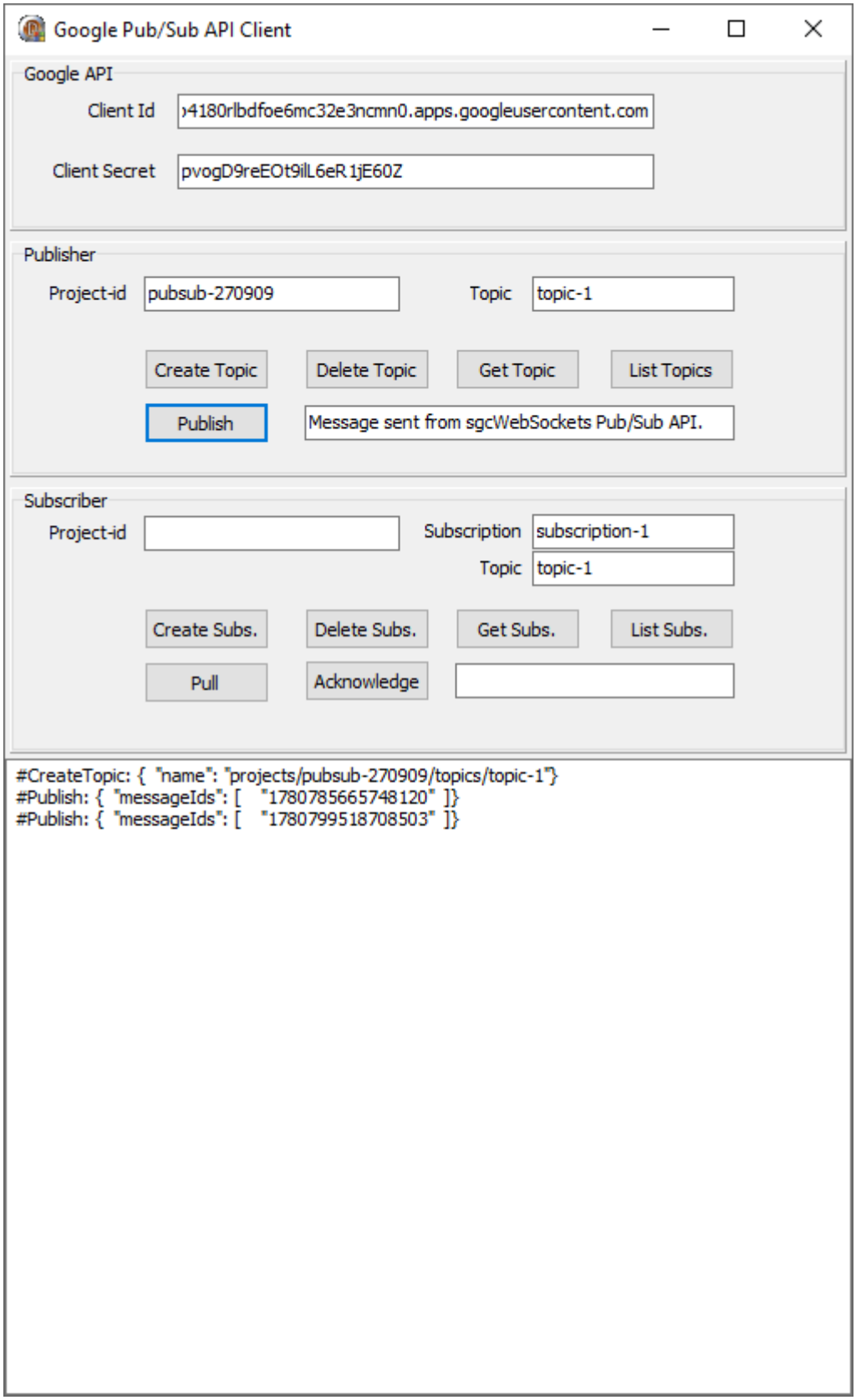
This is how it must be configured in the sgcWebSocket PubSub sample.



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.



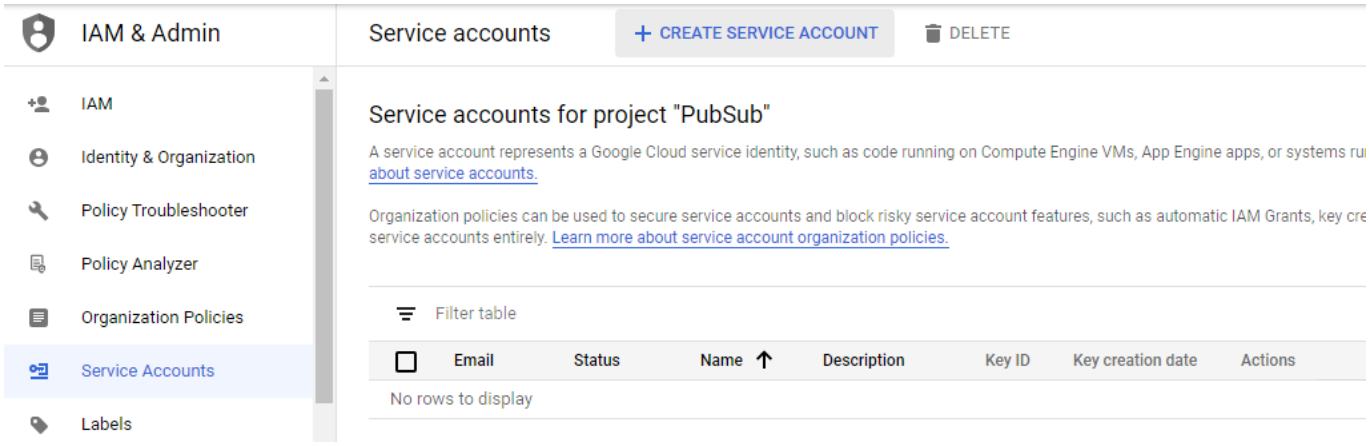
Allow the connection, and if successful, you can start working with this API.


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"]}

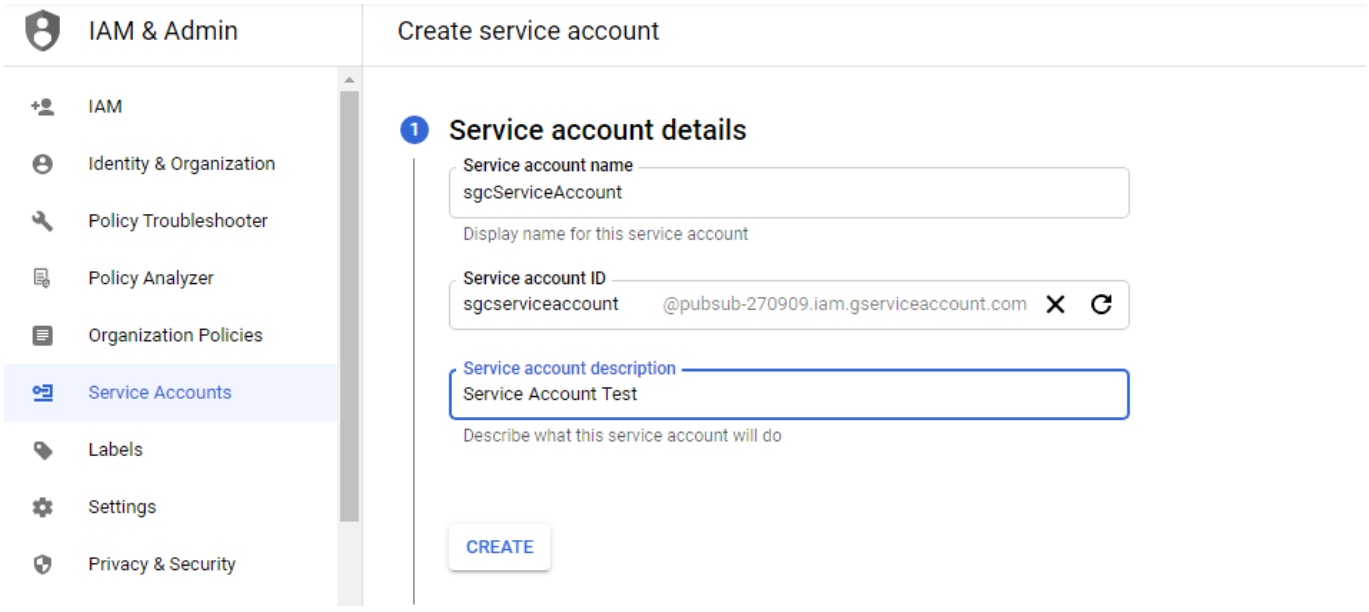
Google Cloud | Google Service Accounts

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.

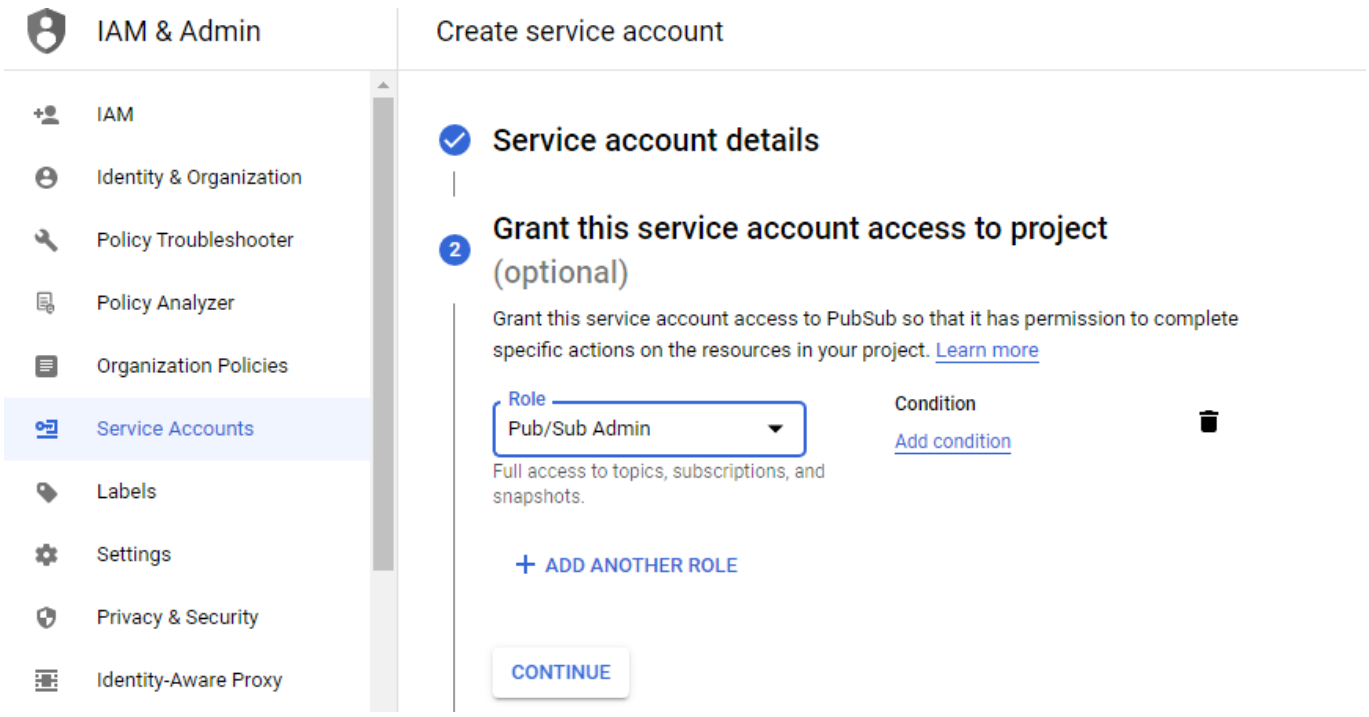
First **login** to your **Google Cloud Account** and use an existing project or create a new one.



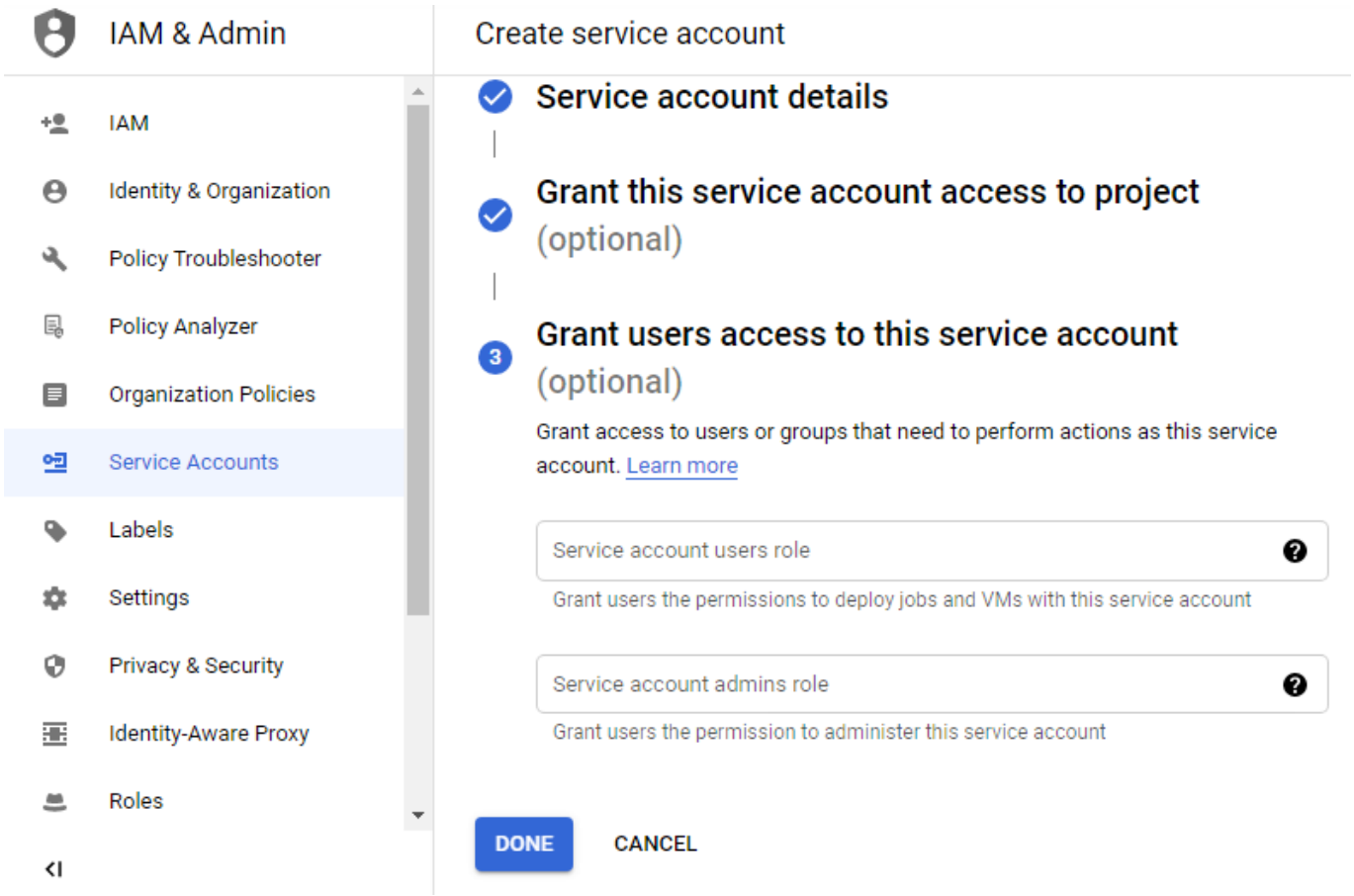
Select **CREATE SERVICE ACCOUNT** and a new page will be shown where you must set the service account name and description



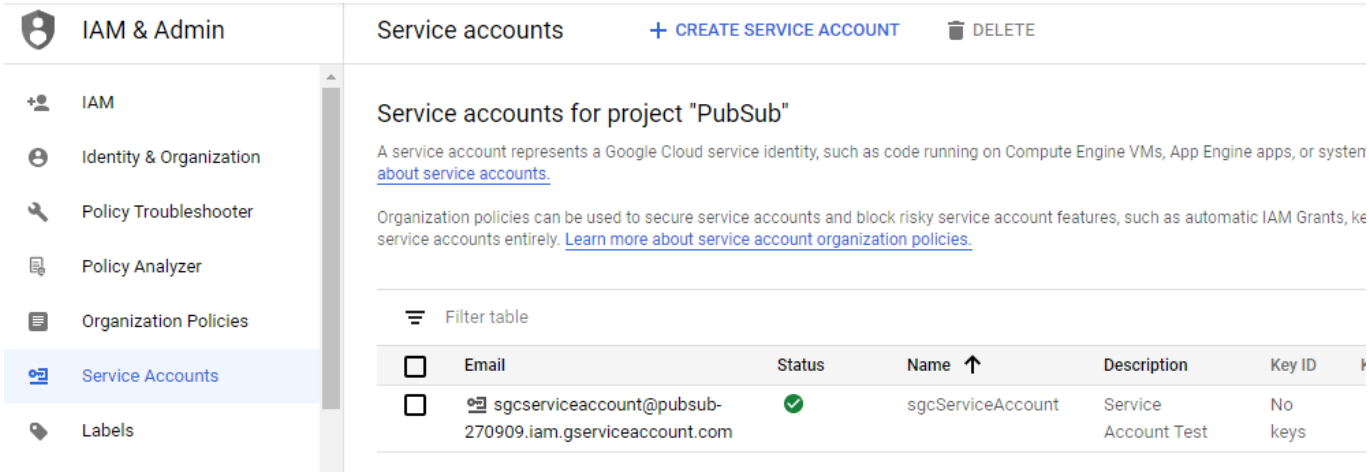
Then select at least one role. Here we select PubSub Admin to allow the client to publish and subscribe to topics, but you can select another role with fewer privileges.



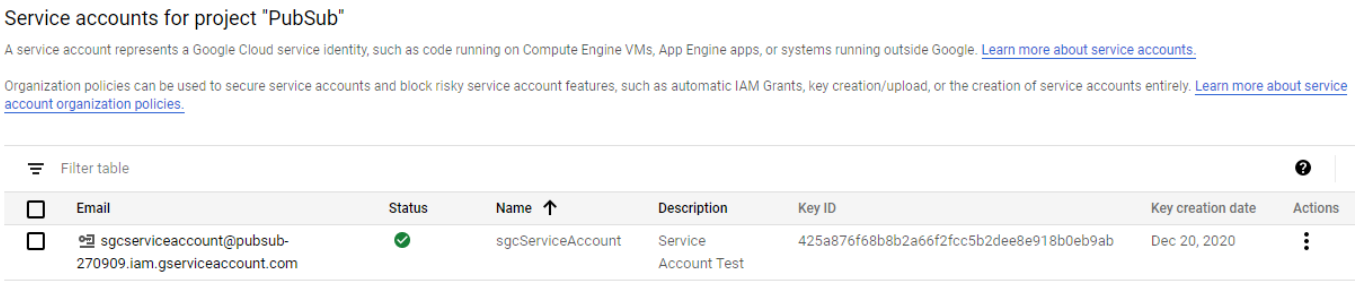
Press CONTINUE and finally you can grant access to other users



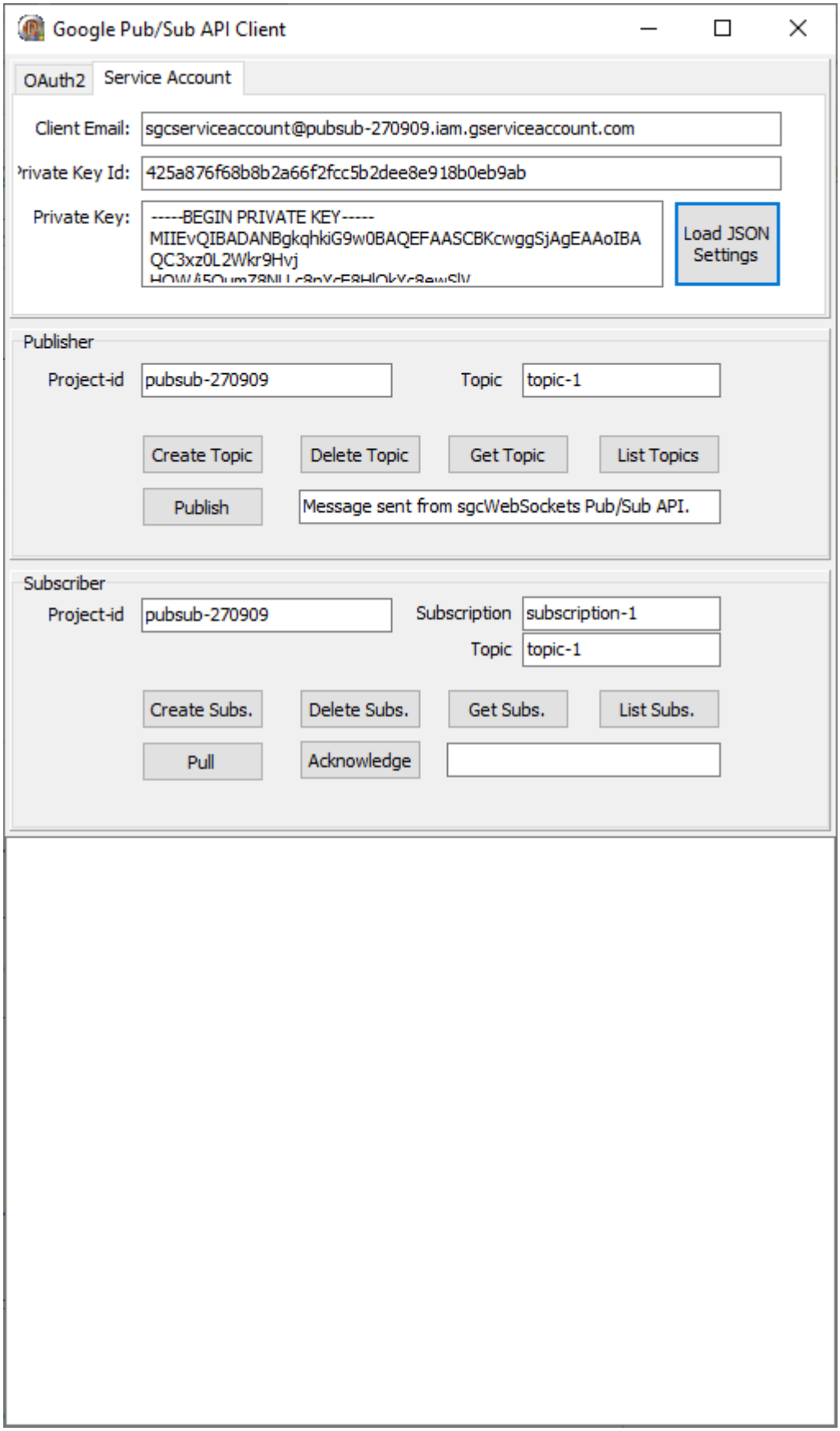
Press DONE when you finish, and a new record will be shown.



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?

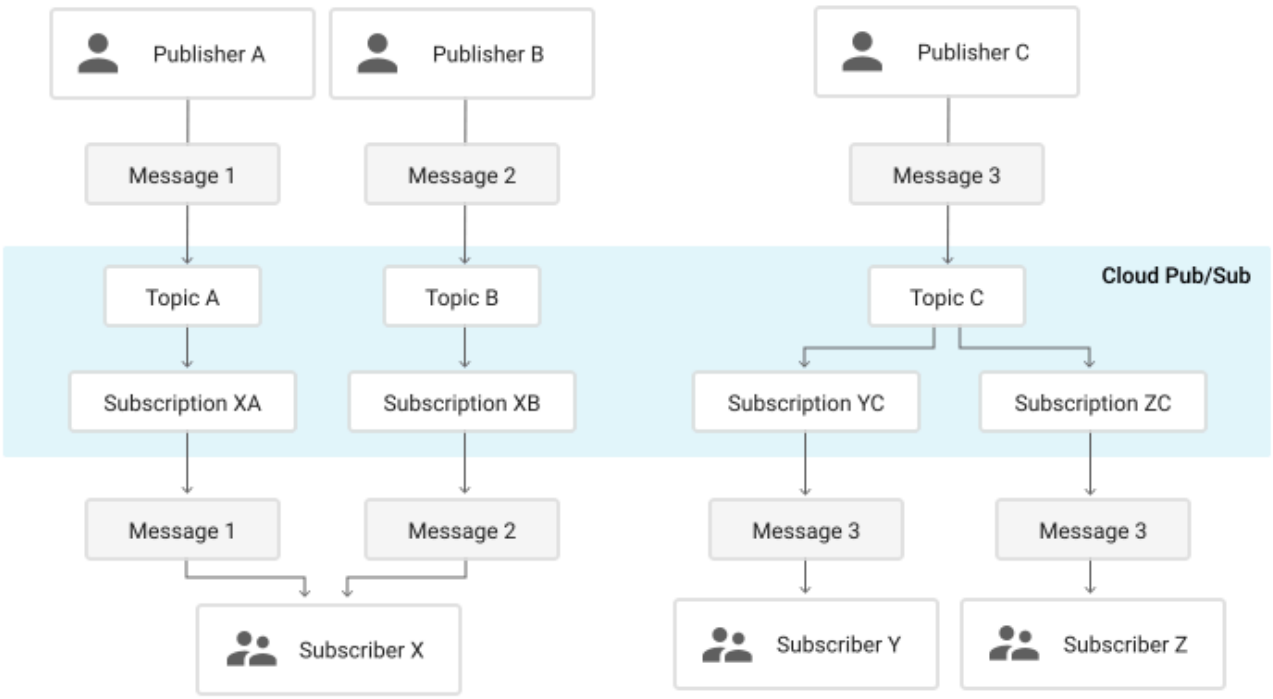
Pub/Sub brings the flexibility and reliability of enterprise message-oriented middleware to the cloud. At the same time, Pub/Sub is a scalable, durable event ingestion and delivery system that serves as a foundation for modern stream analytics pipelines. By providing many-to-many, asynchronous messaging 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:

```
oPubSub := TsgcHTTPGoogleCloud_PubSub_Client.Create(nil);
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 := TsgcHTTPGoogleCloud_PubSub_Client.Create(nil);
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.

```
oPubSub := TsgcHTTPGoogleCloud_PubSub_Client.Create(nil);
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 := TsgcHTTPGoogleCloud_PubSub_Client.Create(nil);
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 := TsgcHTTPGoogleCloud_PubSub_Client.Create(nil);
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

```
oPubSub := TsgcHTTPGoogleCloud_PubSub_Client.Create(nil);
oPubSub.GoogleCloudOptions.OAuth2.ClientId := '... your google client id...';
oPubSub.GoogleCloudOptions.OAuth2.ClientSecret := '... your google client secret...';
oAttributes := TStringList.Create;
Try
  oAttributes.CommaText := 'origin=gcloud-sample,username=gcp';
  oPubSub.Publish('pubsub-270909', 'topic-1', 'My First Message from sgcWebSockets.', oAttributes, 'username'));
Finally
  oAttributes.Free;
end;
```

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 := TsgcHTTPGoogleCloud_PubSub_Client.Create(nil);
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 := TsgcHTTPGoogleCloud_PubSub_Client.Create(nil);
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:

```
oCalendar := TsgcHTTPGoogleCloud_Calendar_Client.Create(nil);
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.

```
oGoogleCalendar := TsgcHTTPGoogleCloud_Calendar_Client.Create(nil);
// ... configure OAuth2 options
oGoogleCalendar.GoogleCloudOptions.OAuth2.ClientId := 'google ClientId';
oGoogleCalendar.GoogleCloudOptions.OAuth2.ClientSecret := 'google ClientSecret';
// ... request calendars
if oGoogleCalendar.LoadCalendars then
begin
  // ... get calendars data
  for i := 0 to oGoogleCalendar.Calendars.Count - 1 do
    vCalendarTitle := oGoogleCalendar.Calendars.Calendar[i].Summary;
end
else
  raise Exception.Create('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.

```
oGoogleCalendar := TsgcHTTPGoogleCloud_Calendar_Client.Create(nil);
// ... 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) then
begin
  // ... get events data
  for i := 0 to oCalendar.Events.Count - 1 do
    vEventTitle := oCalendar.Events[i].Summary;
end
else
  raise Exception.Create('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:

```
procedure OnGoogleCalendarAuthToken(Sender: TObject; const TokenType, Token, Data: string);
var
  oINI: TINIFile;
  oJSON: TsgcJSON;
begin
  oJSON := TsgcJSON.Create(nil);
  Try
    oJSON.Read(Data);
    if oJSON.Node['refresh_token']  nil then
    begin
      oINI := TINIFile.Create(ChangeFileExt(Application.ExeName, '.ini'));
      Try
        oINI.WriteString('OAUTH2', 'Token', oJSON.Node['refresh_token'].Value);
      Finally
        oINI.Free;
      End;
    end;
  Finally
    oJSON.Free;
  End;
end;
```

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:

```
oFCM := TsgcHTTPGoogleCloud_FCM_Client.Create(nil);
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:= TsgcHTTPGoogleCloud_FCM_Client.Create(nil);
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.

```
oFCM := TsgcHTTPGoogleCloud_FCM_Client.Create(nil);
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 creden-tials 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 := TsgcUDPClient.Create(nil);
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 := TsgcUDPServer.Create(nil);
oClient.Port := 80;
```

3. To **start** the server, set the property **Active = true**.

Reference

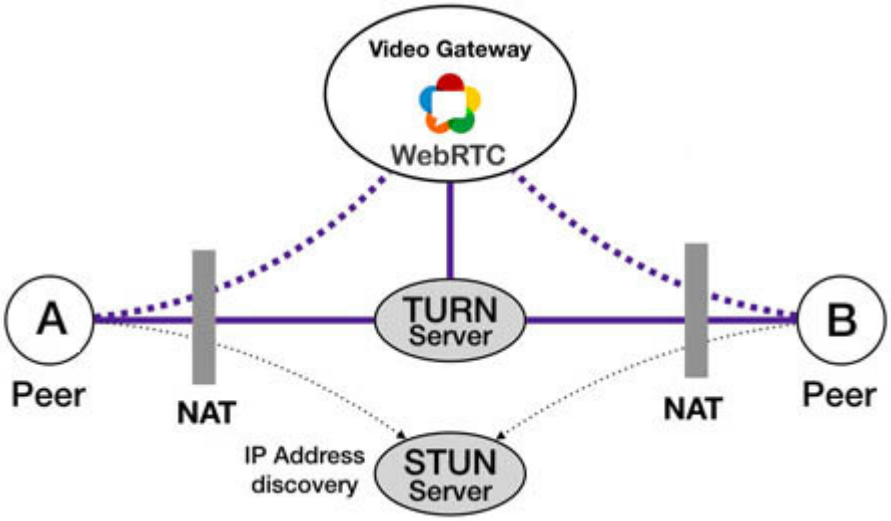
- Properties — 11 properties available on the component.
- Methods — 7 public methods.
- Events — 6 events you can handle.

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.

```
oSTUN := TsgcSTUNClient.Create(nil);
oSTUN.Host := 'stun.sgcwebsockets.com';
oSTUN.Port := 3478;
oSTUN.SendRequest;

procedure OnSTUNResponseSuccess(Sender: TObject; const aSocket: TsgcSocketConnection;
  const aMessage: TsgcSTUN_Message; const aBinding: TsgcSTUN_ResponseBinding);
begin
  DoLog('Remote IP: ' + aBinding.RemoteIP + '. Remote Port: ' + IntToStr(aBinding.RemotePort));
end;

procedure OnSTUNResponseError(Sender: TObject; const aSocket: TsgcSocketConnection;
  const aMessage: TsgcSTUN_Message; const aError: TsgcSTUN_ResponseError);
begin
  DoLog('Error: ' + IntToStr(aError.Code) + ' ' + aError.Reason);
end;
```

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.

```
oSTUN := TsgcSTUNClient.Create(nil);
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.

```
oSTUN := TsgcSTUNClient.Create(nil);
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.

```
procedure OnSTUNResponseSuccess(Sender: TObject; const aSocket: TsgcSocketConnection;
  const aMessage: TsgcSTUN_Message; const aBinding: TsgcSTUN_ResponseBinding);
var
  i: Integer;
begin
  DoLog('#binding: ' + aBinding.RemoteIP + ':' + IntToStr(aBinding.RemotePort));
  for i := 0 to aMessage.Attributes.Count - 1 do
  begin
    case TsgcSTUN_Attribute(aMessage.Attributes.Items[i]).AttributeType of
      stmaFingerprint:
        DoLog('#fingerprint: ' + IntToStr(TsgcSTUN_Attribute_FINGERPRINT
          (aMessage.Attributes.Items[i]).Fingerprint));
      stmaSoftware:
        DoLog('#software: ' + TsgcSTUN_Attribute_SOFTWARE
          (aMessage.Attributes.Items[i]).Software);
      stmaResponse_Origin:
        DoLog('#response_origin: ' + TsgcSTUN_Attribute_RESPONSE_ORIGIN
          (aMessage.Attributes.Items[i]).Address + ':' +
          IntToStr(TsgcSTUN_Attribute_RESPONSE_ORIGIN(aMessage.Attributes.Items
          [i]).Port));
      stmaOther_Address:
        DoLog('#other_address: ' + TsgcSTUN_Attribute_OTHER_ADDRESS
          (aMessage.Attributes.Items[i]).Address + ':' +
          IntToStr(TsgcSTUN_Attribute_OTHER_ADDRESS(aMessage.Attributes.Items
          [i]).Port));
      stmaSource_Address:
        DoLog('#source_address: ' + TsgcSTUN_Attribute_SOURCE_ADDRESS
          (aMessage.Attributes.Items[i]).Address + ':' +
          IntToStr(TsgcSTUN_Attribute_SOURCE_ADDRESS(aMessage.Attributes.Items
          [i]).Port));
      stmaChanged_Address:
        DoLog('#changed_address: ' + TsgcSTUN_Attribute_CHANGED_ADDRESS
          (aMessage.Attributes.Items[i]).Address + ':' +
          IntToStr(TsgcSTUN_Attribute_CHANGED_ADDRESS(aMessage.Attributes.Items
          [i]).Port));
    end;
  end;
end;
```

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.

```
oSTUN := TsgcSTUNServer.Create(nil);
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).

```
oSTUN := TsgcSTUNServer.Create(nil);
oSTUN.Port := 3478;
oSTUN.STUNOptions.Authentication.Enabled := True;
oSTUN.STUNOptions.Authentication.LongTermCredentials.Enabled := True;
oSTUN.STUNOptions.Authentication.LongTermCredentials.Realm := 'sgcWebSockets';
oSTUN.STUNOptions.Authentication.LongTermCredentials.StaleNonce := 600;
oSTUN.Active := True;

procedure OnSTUNRequestAuthorization(Sender: TObject; const aRequest: TsgcSTUN_Message;
  const aUsername, aRealm: string; var Password: string);
begin
  if aUsername = 'my-user' then
    Password := 'my-password';
end;
```

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.

```
oSTUN := TsgcSTUNServer.Create(nil);
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.

```
oTURN := TsgcTURNClient.Create(nil);
oTURN.Host := 'turn.sgcwebsockets.com';
oTURN.Port := 3478;
oTURN.Allocate;

procedure OnTURNAllocate(Sender: TObject; const aSocket: TsgcSocketConnection;
  const aMessage: TsgcSTUN_Message; const aAllocation: TsgcTURN_ResponseAllocation);
begin
  DoLog('Relayed IP: ' + aAllocation.RelayedIP + '. Relayed Port: ' + IntToStr(aAllocation.RelayedPort));
end;

procedure OnSTUNResponseError(Sender: TObject; const aMessage: TsgcSTUN_Message;
const aError: TsgcSTUN_ResponseError);
begin
  DoLog('Error: ' + IntToStr(aError.Code) + ' ' + aError.Reason);
end;
```

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.

```
oTURN := TsgcTURNClient.Create(nil);
oTURN.Host := 'turn.sgcwebsockets.com';
oTURN.Port := 3478;
oTURN.Allocate();

procedure OnTURNAllocate(Sender: TObject; const aSocket: TsgcSocketConnection; const
aMessage: TsgcSTUN_Message; const aAllocation: TsgcTURN_ResponseAllocation);
begin
  DoLog('Relayed IP: ' + aAllocation.RelayedIP + '. Relayed Port: ' + IntToStr(aAllocation.RelayedPort));
end;


procedure OnSTUNResponseError(Sender: TObject; const aMessage: TsgcSTUN_Message;
const aError: TsgcSTUN_ResponseError);
begin
  DoLog('Error: ' + IntToStr(aError.Code) + ' ' + aError.Reason);
end;
```

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');

procedure OnTURNCreatePermission(Sender: TObject; const aSocket: TsgcSocketConnection;
  const aMessage: TsgcSTUN_Message; const aCreatePermission: TsgcTURN_ResponseCreatePermission);
begin
  DoLog('#Create Permission: ' + aCreatePermission.IPAddresses.Text);
end;
```

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');

procedure OnTURNDataIndication(Sender: TObject; const aSocket: TsgcSocketConnection;
  const aMessage: TsgcSTUN_Message; const aDataIndication: TsgcTURN_ResponseDataIndication);
begin
  DoLog('#Data Indication: [' + aDataIndication.PeerIP + ':' + IntToStr(aDataIndication.PeerPort) + '] ' +
    sgcGetStringFromBytes(aDataIndication.Data));
end;
```

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);

procedure OnTURNChannelBind(Sender: TObject; const aSocket: TsgcSocketConnection;
  const aMessage: TsgcSTUN_Message; const aChannelBind: TsgcTURN_ResponseChannelBind);
begin
  DoLog('#Channel Bind: ' + IntToStr(aChannelBind.Channel));
end;
```

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.

```
procedure OnTURNChannelData(Sender: TObject; const aSocket: TsgcSocketConnection;
  const aChannelData: TsgcTURNChannelData);
begin
  DoLog('#Channel Data: [' + IntToStr(aChannelData.ChannelID) + '] ' +
    sgcGetStringFromBytes(aChannelData.Data));
end;
```

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.

```
oTURN := TsgcTURNServer.Create(nil);
oTURN.Port := 3478;
oTURN.TURNOptions.Authentication.Enabled := True;
oTURN.TURNOptions.Authentication.LongTermCredentials.Enabled := True;
oTURN.TURNOptions.Authentication.LongTermCredentials.Realm := 'esegece.com';
oTURN.Active := True;
procedure OnSTUNRequestAuthorization(Sender: TObject; const aRequest: TsgcSTUN_Message;
  const aUsername, aRealm: string; var Password: string);
begin
  if (aUsername = 'user') and (aRealm = 'esegece.com') then
    Password := 'password';
end;
```

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).

```
oTURN := TsgcTURNServer.Create(nil);
oTURN.Port := 3478;
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.Active := True;

procedure OnSTUNRequestAuthorization(Sender: TObject; const aRequest: TsgcSTUN_Message;
  const aUsername, aRealm: string; var Password: string);
begin
  if (aUsername = 'my-user') and (aRealm = 'sgcWebSockets') then
    Password := 'my-password';
end;
```

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.

```
procedure OnTURNBeforeAllocate(Sender: TObject; const aSocket: TsgcSocketConnection;
  const aIP: string; aPort: Word; var Reject: Boolean);
begin
  if not (your own rules) then
        Reject := false;
end;
```

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.

```
procedure OnTURNCreateAllocation(Sender: TObject; const aSocket: TsgcSocketConnection;
  const Allocation: TsgcTURNAllocationItem);
begin
  DoLog('New Allocation: ' + Allocation.RelayIP + ':' + IntToStr(Allocation.RelayPort));
end;
```

When the Allocation expires or is deleted receiving a Refresh Request from client with a lifetime of zero, the event **OnTURNDeleteAllocation** event is fired.

```
procedure OnTURNDeleteAllocation(Sender: TObject; const aSocket: TsgcSocketConnection;
  const Allocation: TsgcTURNAllocationItem);
begin
  DoLog('Allocation Deleted: ' + Allocation.RelayIP + ':' + IntToStr(Allocation.RelayPort));
end;
```

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.

```
oICE := TsgcICEClient.Create(nil);
oTURN := TsgcTURNClient.Create(nil);
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.

```
oICE := TsgcICEClient.Create(nil);
oTURN := TsgcTURNClient.Create(nil);
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();

procedure OnICECandidate(Sender: TObject; const aCandidate: TsgcICE_Candidate);
begin
  DoLog('[#Candidate] ' + aCandidate.AsString);
end;
```

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.

```
oServer := TsgcWebSocketServer.Create(nil);
oProtocol := TsgcWSPServer_RTCPeerConnection.Create(nil);
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+)

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 root certificate file.
CertFile: path to certificate file.
KeyFile: path to certificate key file.
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
OpenSSL_Options: configuration of the openSSL libraries.
- **APIVersion:** allows defining which OpenSSL API will be used. only openSSL API 1.1+ supports DTLS.
  - **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.

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.

```
oRTCPeerConnection := TsgcRTCPeerConnection.Create(nil);
...
oRTCPeerConnection.WriteData('Hello from sgcWebSockets!!!');
```

Receive Data

Every time the TsgcRTCPeerConnection receives any data from the other peer, the event OnRTCMessage will be called.

```
procedure OnRTCMessage(Sender: TObject; const aBytes: TBytes);
begin
  ShowMessage(TEncoding.UTF8.GetString(aBytes));
end;
```

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 := TsgcWSHTTPWebBrokerBridgeServer.Create(Self);
FServer.OnCommandRequest := OnCommandRequestEvent;
FServer.OnCommandGet := OnCommandGetevent;

procedure OnCommandRequestEvent(AThread: TIdContext; ARequestInfo: TIdHTTPRequestInfo;
AResponseInfo: TIdHTTPResponseInfo; var aHandled: Boolean);
begin
  if ARequestInfo.Document = '/test.html' then
    aHandled := True;
end;

procedure OnCommandGetevent(AContext: TIdContext; ARequestInfo: TIdHTTPRequestInfo;
AResponseInfo: TIdHTTPResponseInfo);
begin
  if ARequestInfo.Document = '/test.html' then
  begin
    AResponseInfo.ResponseNo := 200;
    AResponseInfo.ContentText := 'hello all';
  end;
end;
```

Load Balancer

If the server is behind the TsgcWebSocketLoadBalancerServer, you may have issues with CORS, to avoid these issues, use the following code

```
procedure TWebModule1.WebModuleBeforeDispatch(Sender: TObject; Request: TWebRequest; Response: TWebResponse; var
begin
  Response.SetCustomHeader('Access-Control-Allow-Origin','*');
  if Trim(Request.GetFieldByName('Access-Control-Request-Headers')) <> '' then
```

```
  begin
    Response.SetCustomHeader('Access-Control-Allow-Headers', Request.GetFieldByName('Access-Control-Request-Heade
    Handled := True;
  end;
  if FServerFunctionInvokerAction <> nil then
    FServerFunctionInvokerAction.Enabled := AllowServerFunctionInvoker;
end;
```

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 := TsgcWSHTTP2WebBrokerBridgeServer.Create(Self);
FServer.OnCommandRequest := OnCommandRequestEvent;
FServer.OnCommandGet := OnCommandGetevent;

procedure OnCommandRequestEvent(AThread: TIdContext; ARequestInfo: TIdHTTPRequestInfo;
AResponseInfo: TIdHTTPResponseInfo; var aHandled: Boolean);
begin
  if ARequestInfo.Document = '/test.html' then
    aHandled := True;
end;

procedure OnCommandGetevent(AContext: TIdContext; ARequestInfo: TIdHTTPRequestInfo;
AResponseInfo: TIdHTTPResponseInfo);
begin
  if ARequestInfo.Document = '/test.html' then
  begin
    AResponseInfo.ResponseNo := 200;
    AResponseInfo.ContentText := 'hello all';
  end;
end;
```

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

```
FServer := TsgcWSServer_HTTPAPI_WebBrokerBridge.Create(Self);
FServer.OnCommandRequest := OnCommandRequestEvent;
FServer.OnMessage := OnWebSocketMessage;

procedure OnCommandRequestEvent(aConnection : TsgcWSConnection_HTTPAPI;
  const aRequestInfo: THttpServerRequest; var aResponseInfo: THttpServerResponse; var aHandled: Boolean);
begin
  if ARequestInfo.Document = '/test.html' then
  begin
    AResponseInfo.ResponseNo := 200;
    AResponseInfo.ContentText := '... body ...';
    aHandled := True;
  end;
end;
procedure OnWebSocketMessage(aConnection: TsgcWSConnection; const aText: string);
begin
  aConnection.WriteData(aText);
end;
```

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)
- **Swagger 1.*** (automatically converted from 1.0 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.

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** Format and creates automatically a **Delphi Client in Native Pascal Code**.

The **sgcOpenAPI Parser** is compatible with the following specifications:

- **OpenAPI 3.***
- **Swagger 2.*** (automatically converted from 2.0 to 3.0)
- **Swagger 1.*** (automatically converted from 1.0 to 3.0)

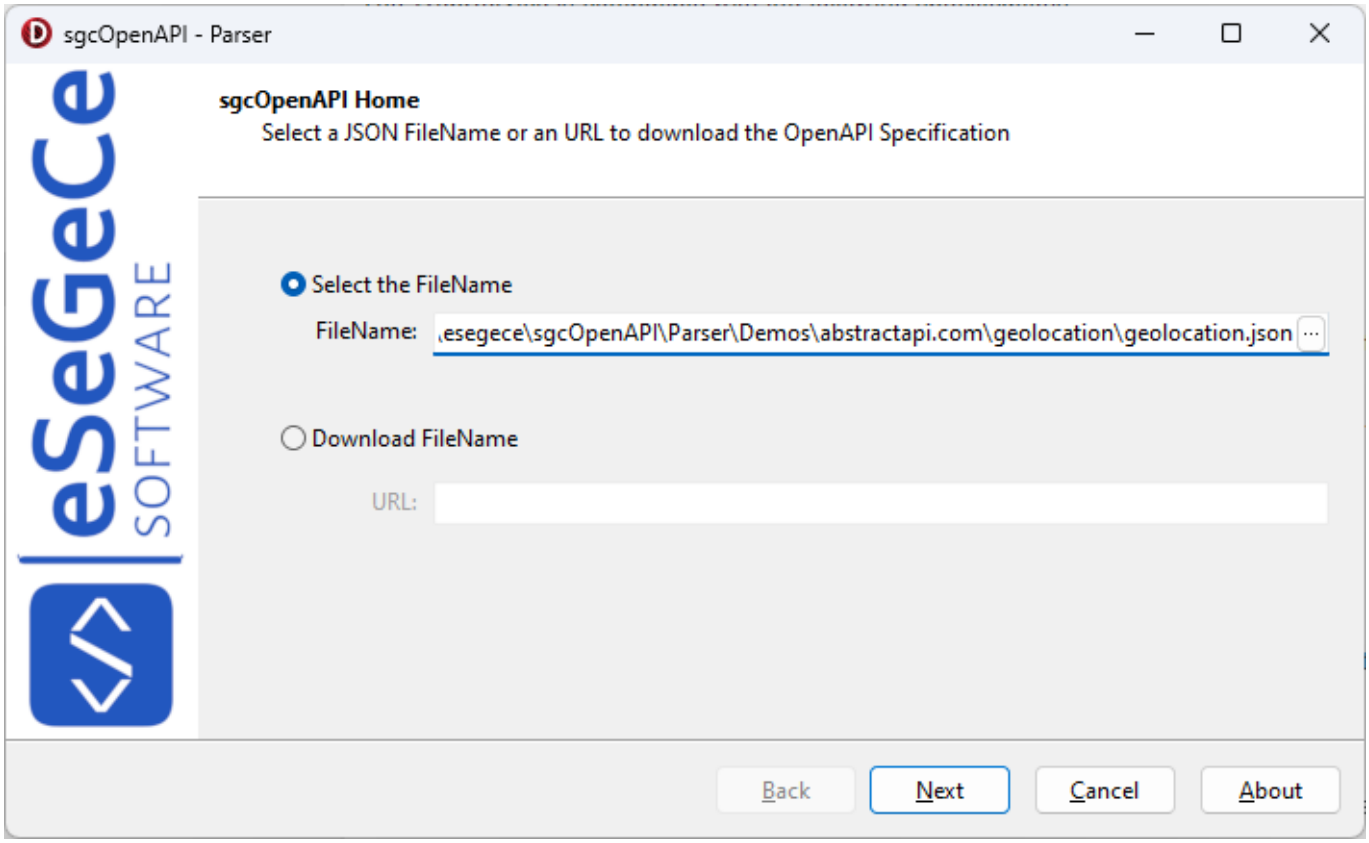
The specification file must be in **JSON** format, if the specification is in YAML format it will be converted automatically. The parser allows you to import specifications split into **multiple schemas**, if you provide an URL it will try to download the specifications and if you provide a filename it will try to load the external schemas from your local drive.

The OpenAPI Parser executable can handle command line parameters or use as a library.

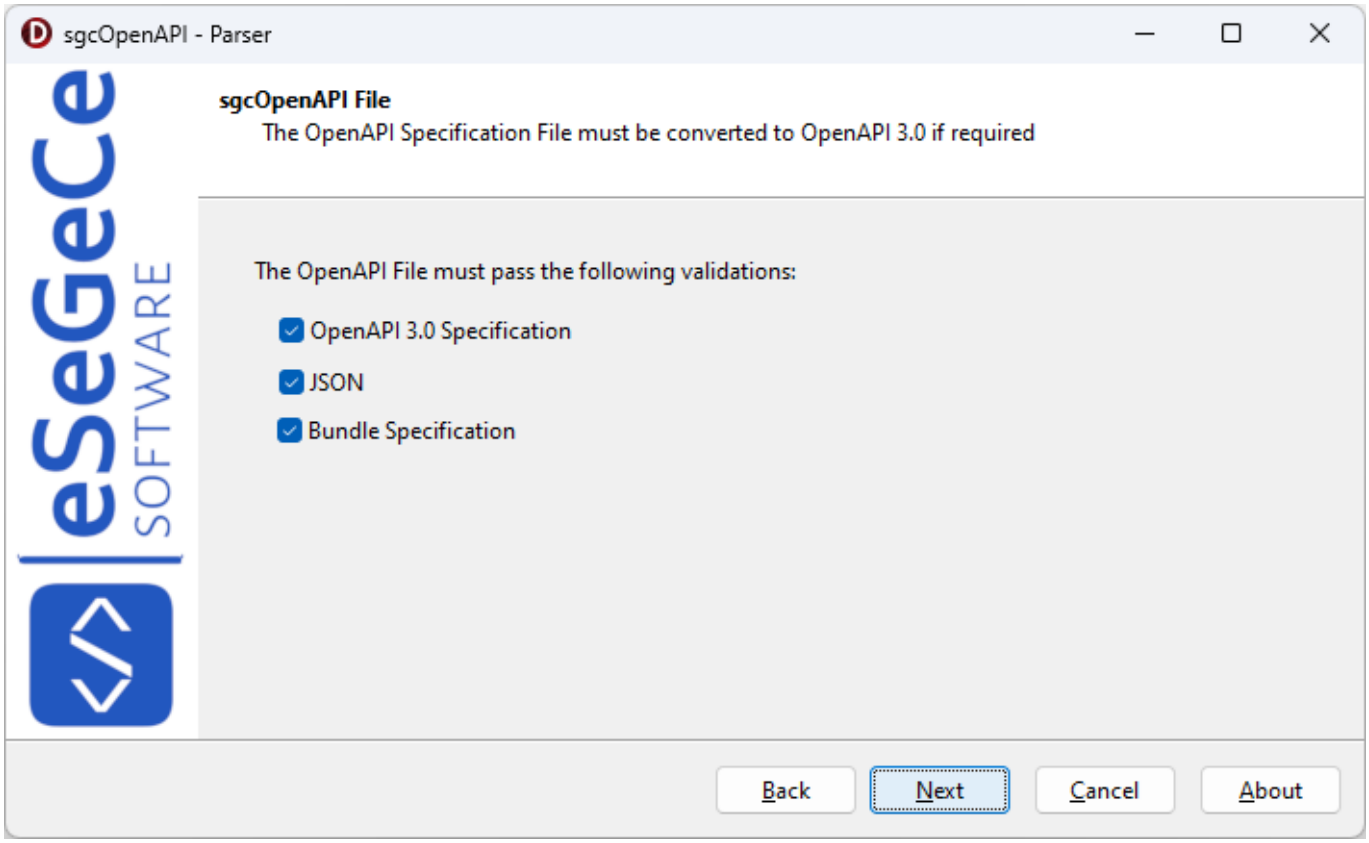
Importing OpenAPI Specification

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:

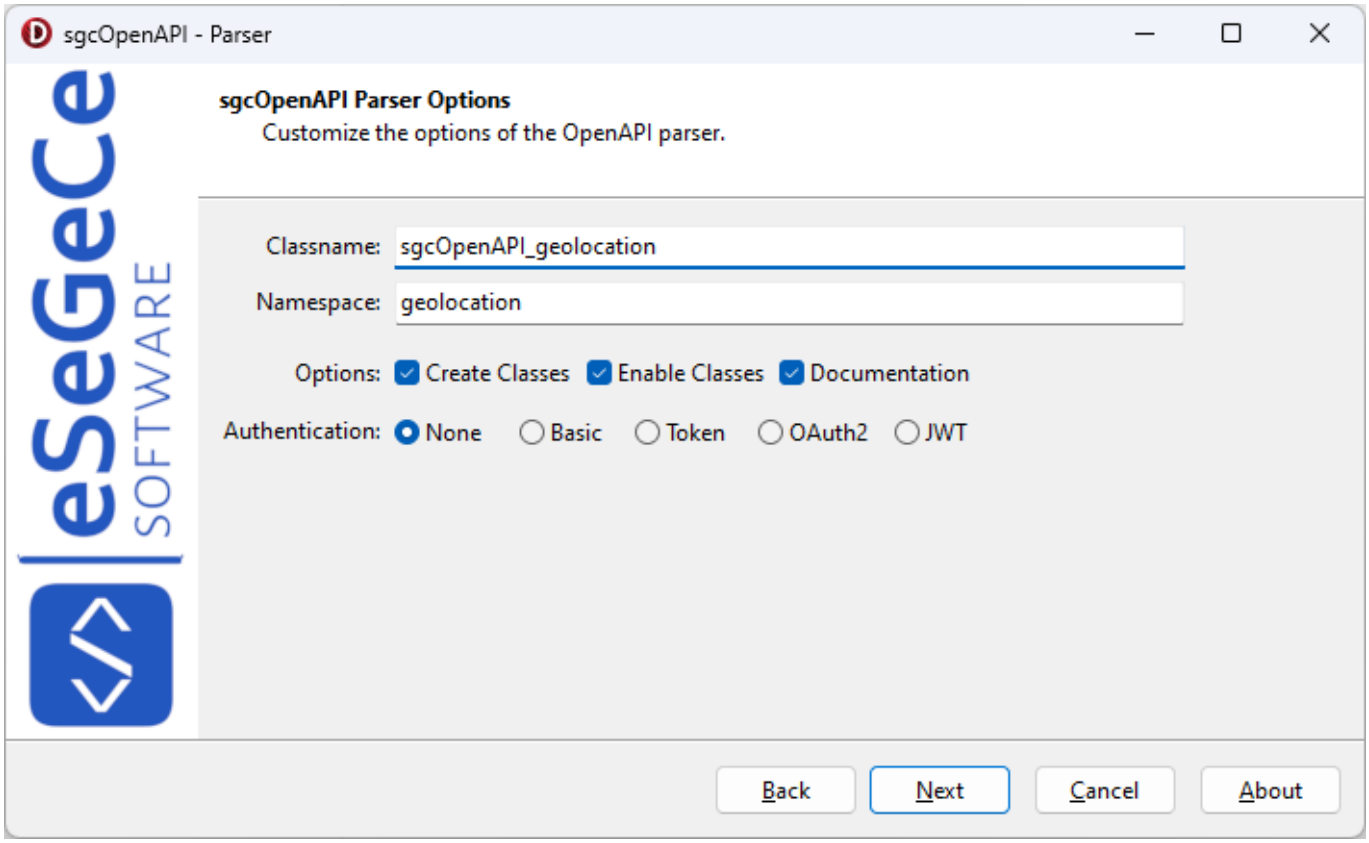
- **Execute the sgcOpenAPI Parser** and select the specification to import or the URL to download.



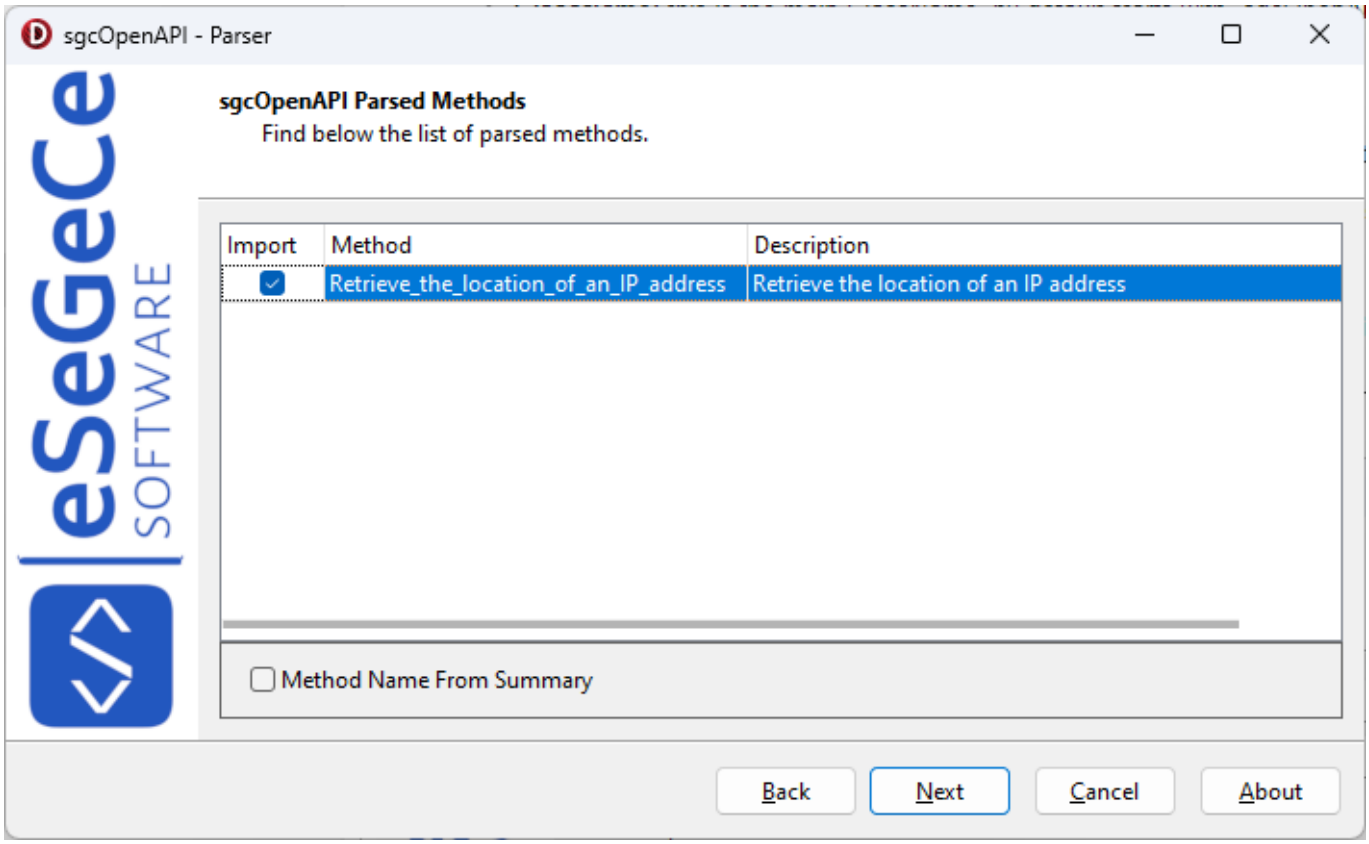
- The **specification is verified** and if it's compatible with the Parser you can continue to the next steps. If the specification makes use of an old version like swagger 2.0 it will be converted automatically. If the specification is split into multiple schemas, the parser will try to bundle a single specification. If there is any problem while converting the file, an error message will appear.



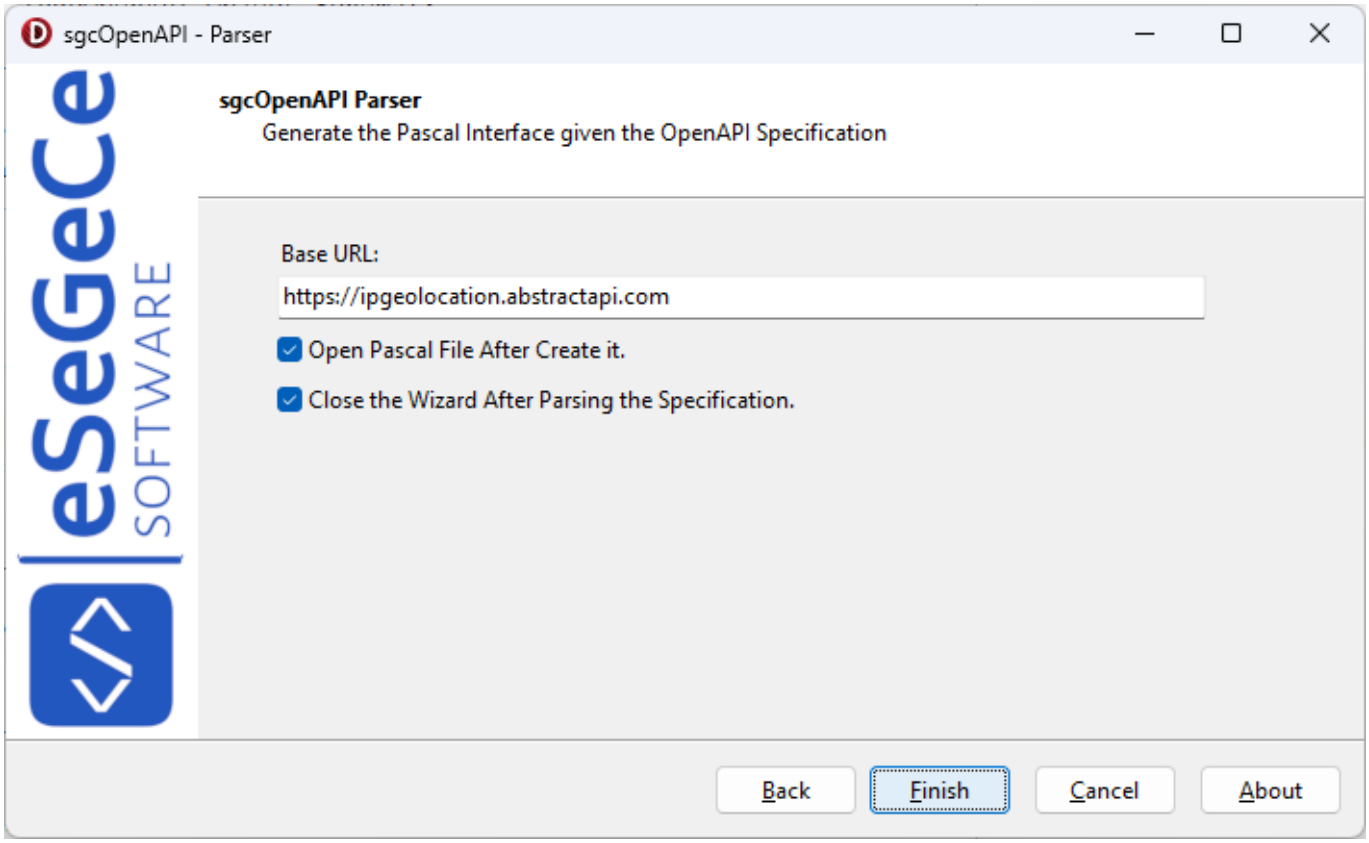
- Now you can **customize the following options** before parsing the document
  - **ClassName:** this is the main ClassName, by default starts with "sgcOpenAPI_" and adds the name of the specification filename.
  - **Namespace:** this is the name of the pascal file generated, by default is the same name 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.

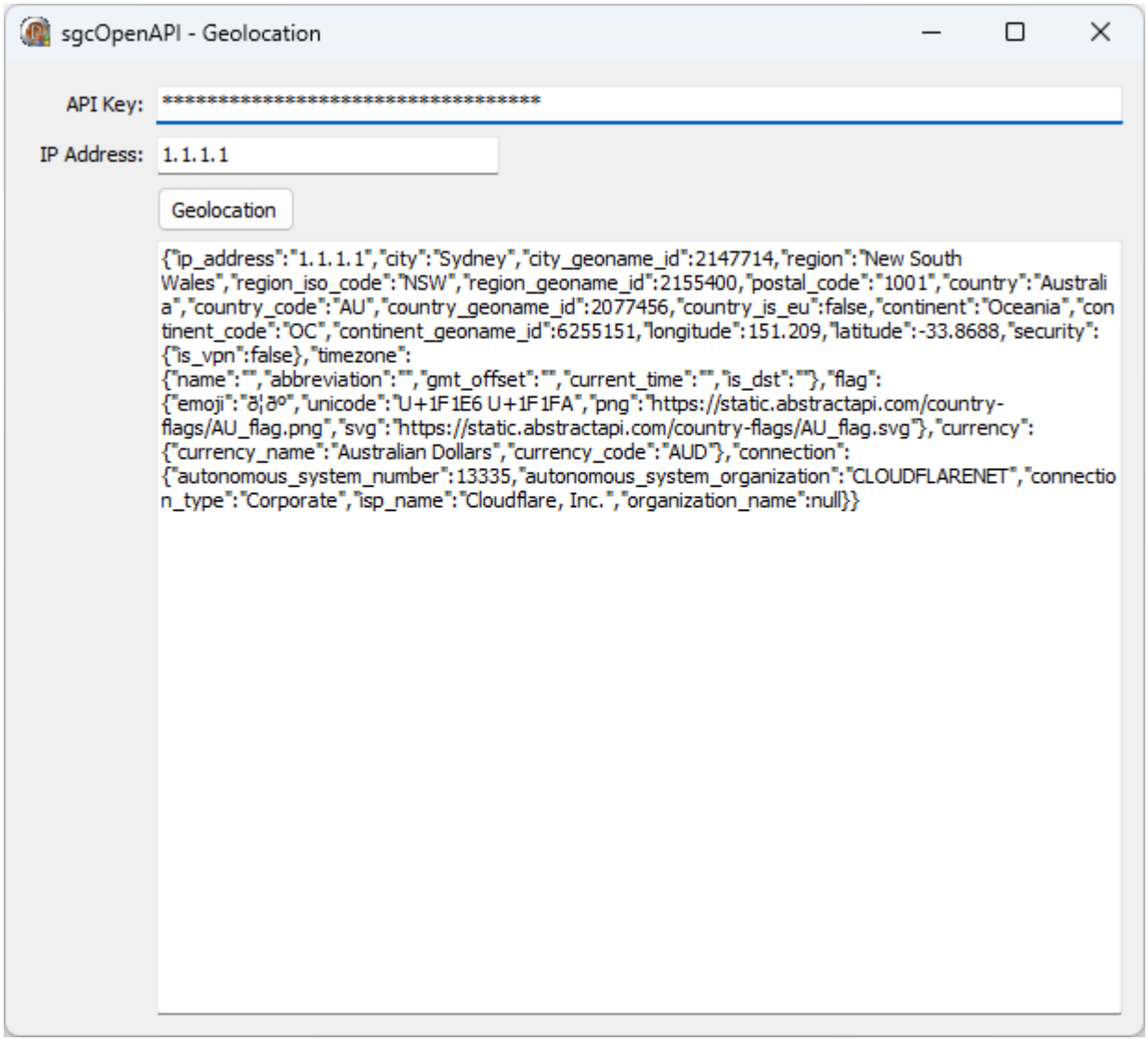
Example

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.


sgcOpenAPI - Geolocation
API Key:
****
IP Address:
1.1.1.1
Geolocation
{"ip_address":"1.1.1.1","city":"Sydney","city_geoname_id":2147714,"region":"New South Wales","region_iso_code":"NSW","region_geoname_id":2155400,"postal_code":"1001","country":"Australia","country_code":"AU","country_geoname_id":2077456,"country_is_eu":false,"continent":"Oceania","continent_code":"OC","continent_geoname_id":6255151,"longitude":151.209,"latitude":-33.8688,"security":{"is_vpn":false},"timezone":{"name":"","abbreviation":"","gmt_offset":"","current_time":"","is_dst":""},"flag":{"emoji":"ð¦ðº","unicode":"U+1F1E6 U+1F1FA","png":"https://static.abstractapi.com/country-flags/AU_flag.png","svg":"https://static.abstractapi.com/country-flags/AU_flag.svg"},"currency":{"currency_name":"Australian Dollars","currency_code":"AUD"},"connection":{"autonomous_system_number":13335,"autonomous_system_organization":"CLOUDFLARENET","connection_type":"Corporate","isp_name":"Cloudflare, Inc.","organization_name":null}}

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:

```
oClass := TMyClass.Create;
oClass.Property1 := 'value1';
oAdditionalProperties := TCustomAdditionalProperties.Create;
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"
```

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. If the input is YAML or Swagger 1/2 it will be converted automatically to OpenAPI 3 JSON.
- **-o -output** : **[mandatory]** Output Pascal file to generate (Delphi client).
- **-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.
- **-l -log** : **[optional]** Show log output while parsing.
- **-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
```

Notes

- If the input specification is not valid or is not a bundled OpenAPI 3 JSON file, the parser will attempt to convert and bundle it automatically before generation.
- When using URLs, the parser downloads the specification before validating and generating the output.

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).

```
function CreateOpenAPIPascalFile(const aInputFile, aOutputFile: PWideChar): Boolean; stdcall;
```

Parameters

- **aInputFile:** full path (or URL) of the OpenAPI specification to parse. If the file is YAML or Swagger 1/2, the parser will normalize it to OpenAPI 3 JSON before generating the output.
- **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

```
if not CreateOpenAPIPascalFile('c:\specs\openapi.json', 'c:\clients\orders.pas') then
  ShowMessage(GetOpenAPILastErrors);
```

CreateOpenAPIPascalFileEx

Generates a Pascal client file with explicit control over parser options.

```
function CreateOpenAPIPascalFileEx(const aInputFile, aOutputFile: PWideChar;
  aGenerateClasses, aGenerateJSON, aGenerateDocumentation: Boolean;
  aMethodsName, aAuthentication: Integer; const aBaseURL: PWideChar;
  aEnableLog: Boolean): Boolean; stdcall;
```

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

```
if not CreateOpenAPIPascalFileEx(
  'c:\specs\openapi.json',
  'c:\clients\orders.pas',
  True, True, True,
  0, 3,
  'https://api.example.com/v2',
  True) then
  ShowMessage(GetOpenAPILastErrors);
```

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**.

```
function GetOpenAPILastErrors(var Errors: PWideChar): Integer; stdcall;
```

Return value
Returns the count of errors

Example

```
if not CreateOpenAPIPascalFile('c:\specs\openapi.yaml', 'c:\clients\orders.pas') then
  GetOpenAPILastErrors(vErrors);
```

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:

```
unit sgcOpenAPI_API_Client;
interface
uses
  Classes, SysUtils;
type
  TsgcOpenAPIPascalSectionType = (soapistHeaderStart, soapistHeaderEnums,
    soapistHeaderClasses, soapistHeaderMainClass, soapistHeaderClassesStructure,
    soapistHeaderMainInstance, soapistHeaderEnd, soapistImplementationStart,
    soapistImplementationClasses, soapistImplementationEnd);
procedure sgcOpenAPI_CreateSection(aType: Integer;
  out aValue: PWideChar); stdcall;
implementation
procedure sgcOpenAPI_CreateSection(aType: Integer; out aValue: PWideChar);
begin
  case TsgcOpenAPIPascalSectionType(aType) of
    soapistHeaderStart: ;
    soapistHeaderEnums: ;
    soapistHeaderClasses: ;
    soapistHeaderMainClass: ;
    soapistHeaderClassesStructure: ;
    soapistHeaderMainInstance: ;
    soapistHeaderEnd: ;
    soapistImplementationStart: ;
    soapistImplementationClasses: ;
    soapistImplementationEnd: ;
  end;
end;
exports sgcOpenAPI_CreateSection;
end.
```

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

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 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.
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.

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 JSON body is encoded as UTF8 (by default false).

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.

```
procedure OnBeforeRequestEvent(Sender: TObject; const aRequest: TsgcOpenAPIRequest);
var
  i: Integer;
  oParameter: TsgcOpenAPIParameter;
begin
  for i := 0 to aRequest.Parameters.Count - 1 do
  begin
    oParameter := aRequest.Parameters[i];
    if oParameter._Name = 'meta-modified-from' then
      oParameter._name := 'eventDateTime-from';
    if oParameter._Name = 'meta-modified-to' then
      oParameter._name := 'eventDateTime-to';
  end;
end;
```

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 | 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';
ShowMessage(GetOpenAPIClient.ListBuckets());
```

GetObject

```
GetOpenAPIClient.AmazonOptions.AccessKey := 'AKIAIOSFODNN7EXAMPLE';
GetOpenAPIClient.AmazonOptions.SecretKey := 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY';
ShowMessage(GetOpenAPIClient.GetObject('bucket_name'));
```

PutObject

```
GetOpenAPIClient.AmazonOptions.AccessKey := 'AKIAIOSFODNN7EXAMPLE';
GetOpenAPIClient.AmazonOptions.SecretKey := 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY';
ShowMessage(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

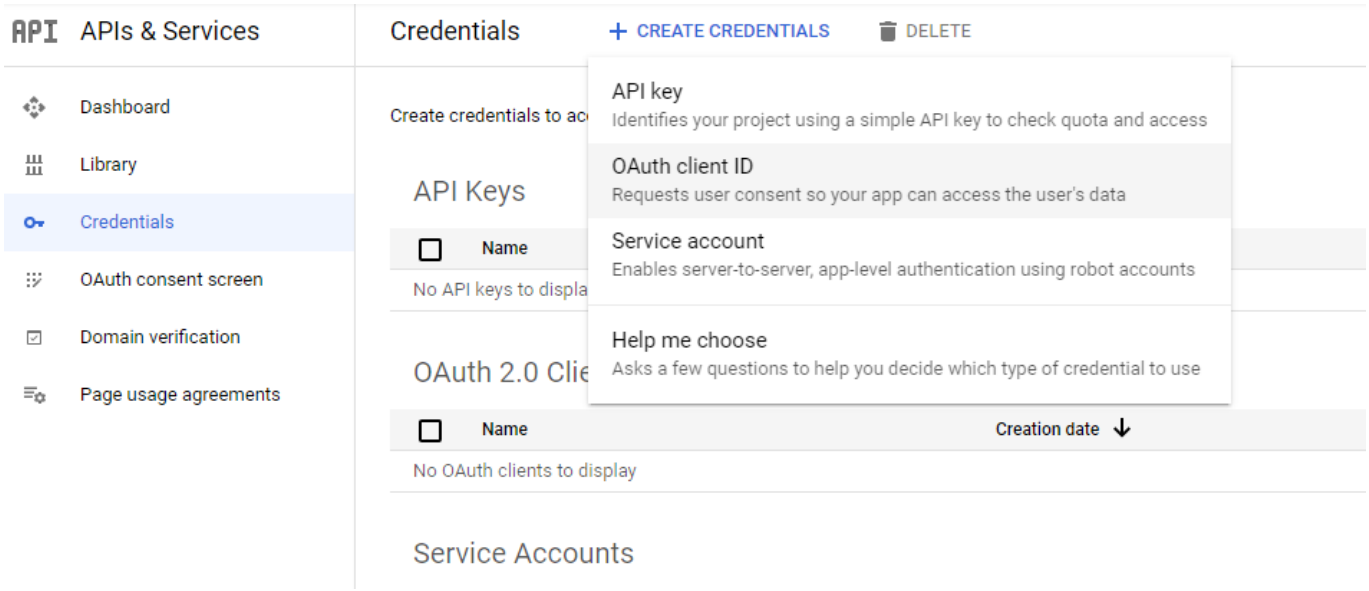
- Google Chat API
- Google Civic Information API
- Google Classroom API
- Google Cloud Data Catalog API
- Google Cloud Deploy API
- Google Cloud Memorystore for Redis API
- Google Cloud Support API
- Google Docs API
- Google Forms API
- Google Identity Toolkit API
- Google Keep API
- Google Mirror
- 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

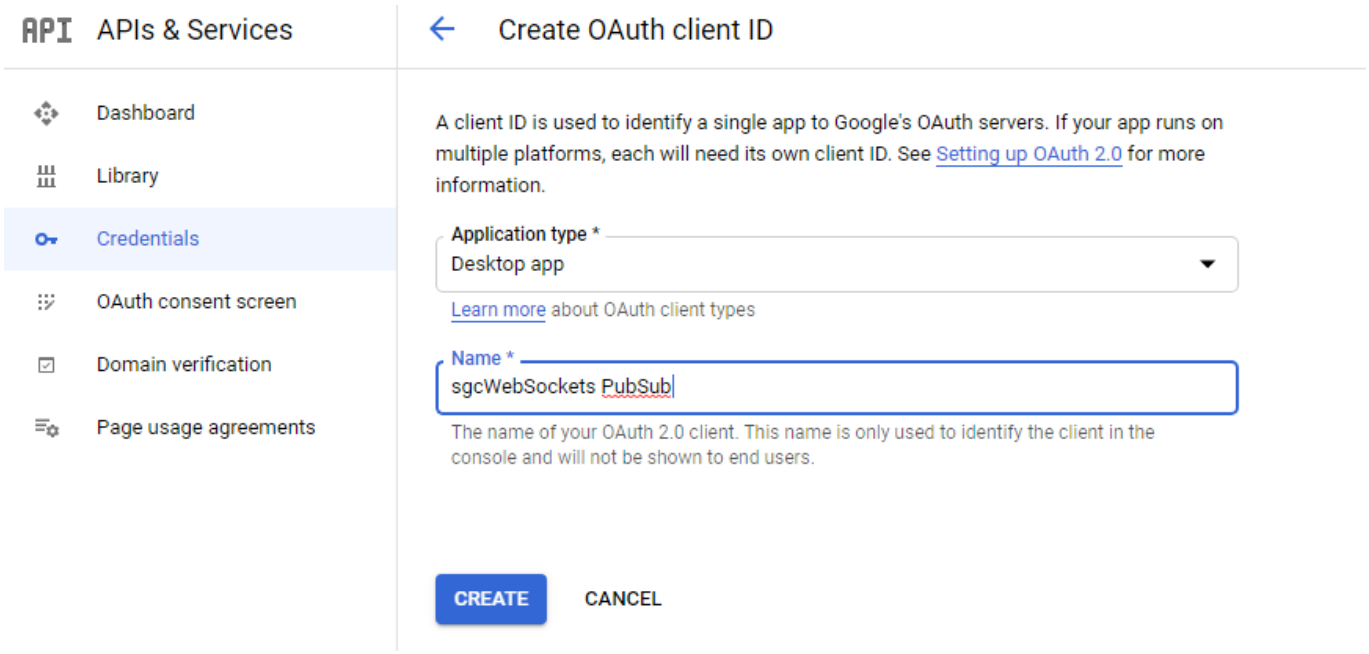
OpenAPI Google Cloud | OAuth2

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.

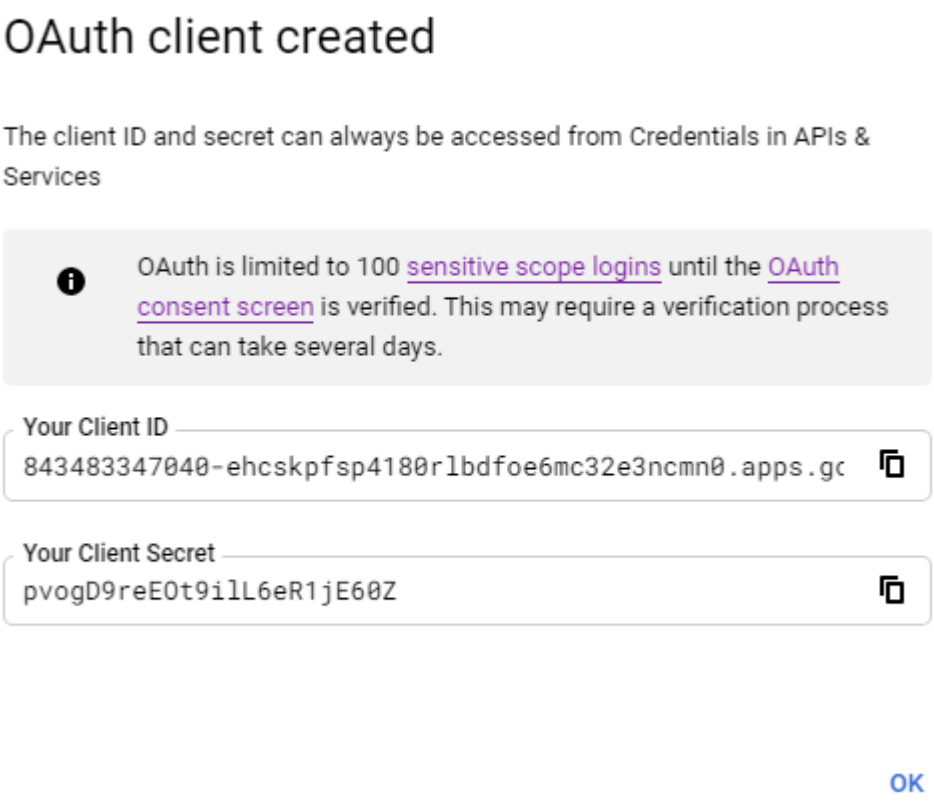
First **login** to your **Google Cloud Account** and use an existing project or create a new one.
After that, go to **Credentials** menu and press the button **CREATE CREDENTIALS**, select the option **OAuth Client ID**.



Select your application type and set a description name

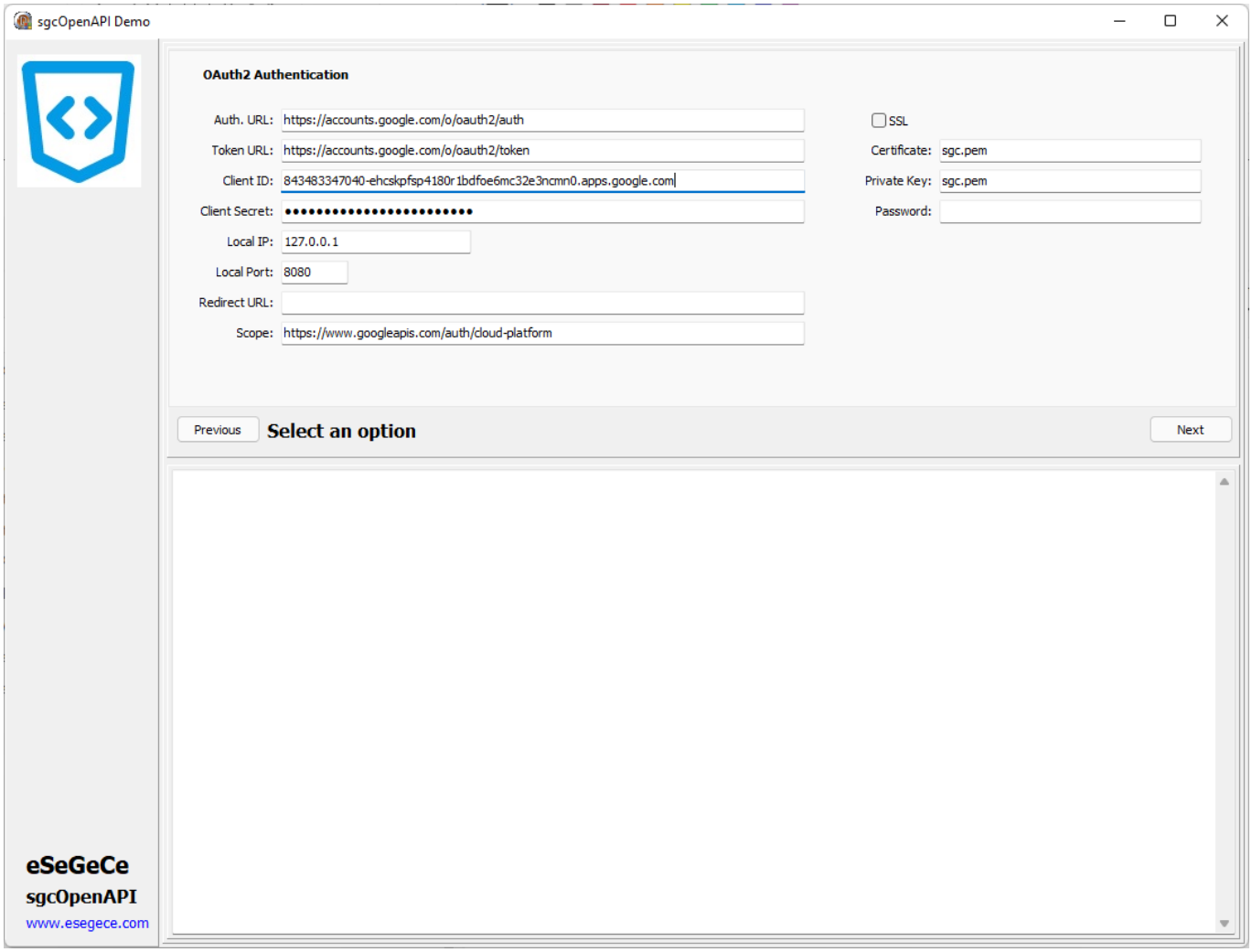


If successful, you will get your **Client Id** and **Client Secret**.



Don't share your OAuth2 data with anyone!

Now copy to the OpenAPI Google Cloud sample



Read more about OAuth2 Configuration.

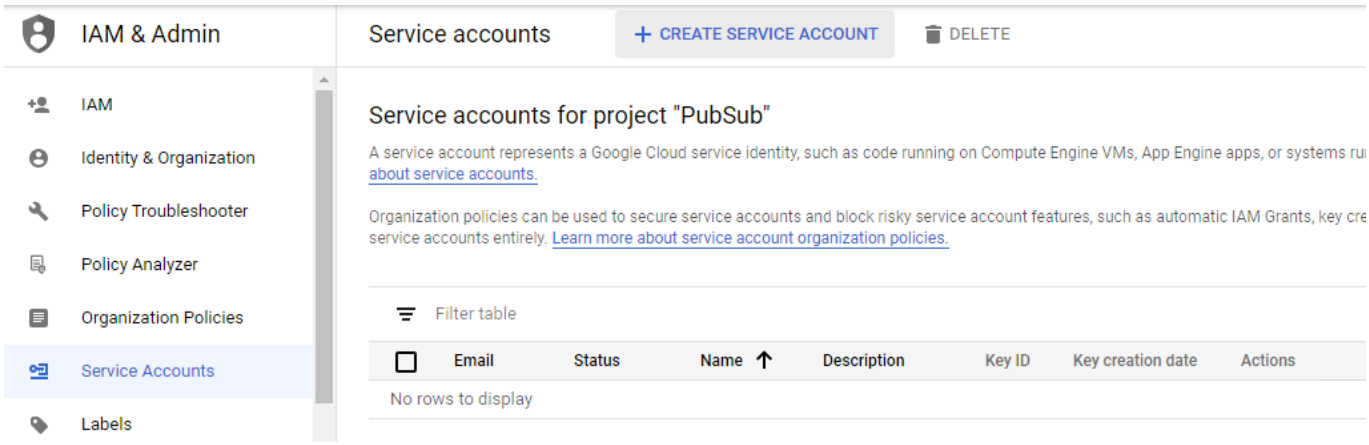
Once you are authenticated, you can **re-authenticate** calling first the method **ClearOAuth2Token** (clear all internal OAuth2 Tokens) and then call any OpenAPI requests, a new web-browser will be shown to re-authenticate against google servers.

OpenAPI Google Cloud | Service Accounts

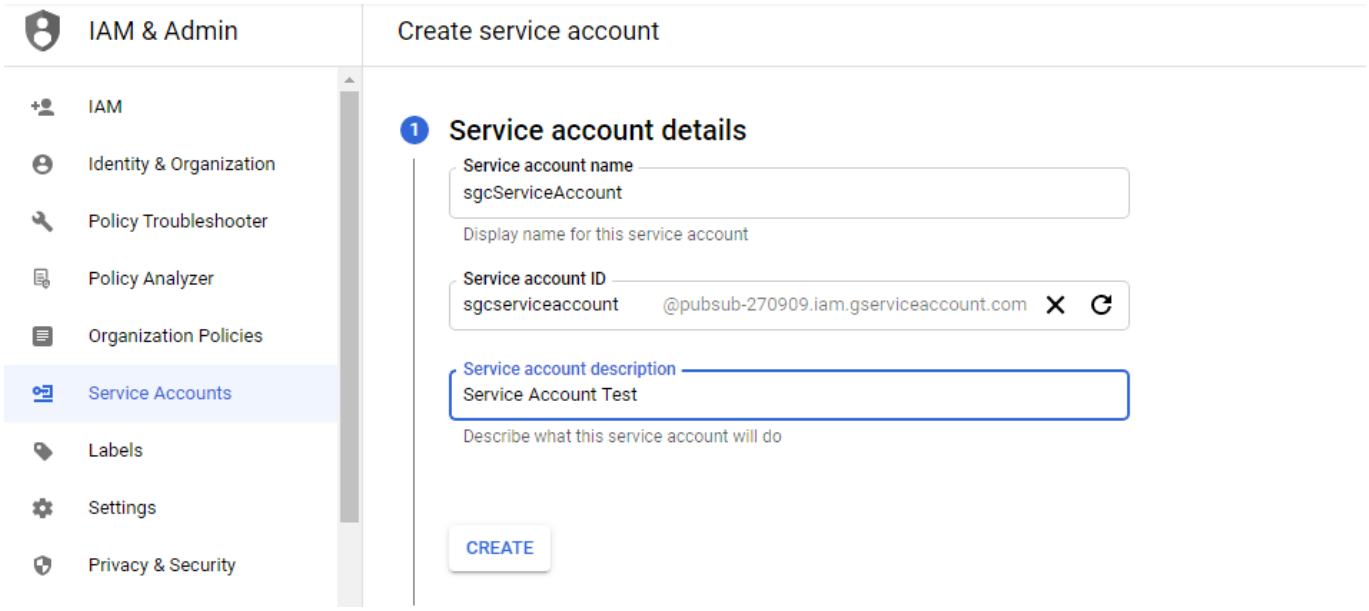
In order to use the **OpenAPI Google Cloud components** and **Authenticate using Service Accounts**, first you must obtain the Private Key Certificate from Google Cloud.

Find below the steps to get Google Private Key Certificate and how to configure in our sample application.

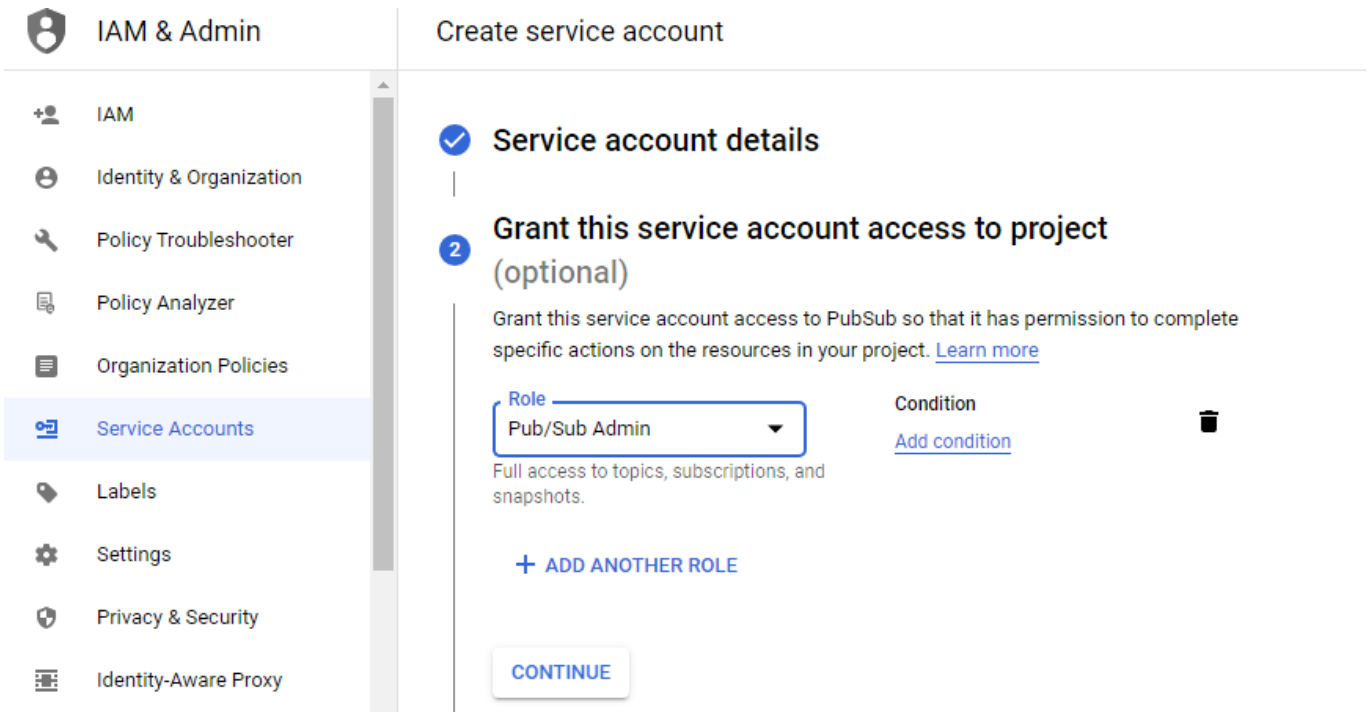
First **login** to your **Google Cloud Account** and use an existing project or create a new one.



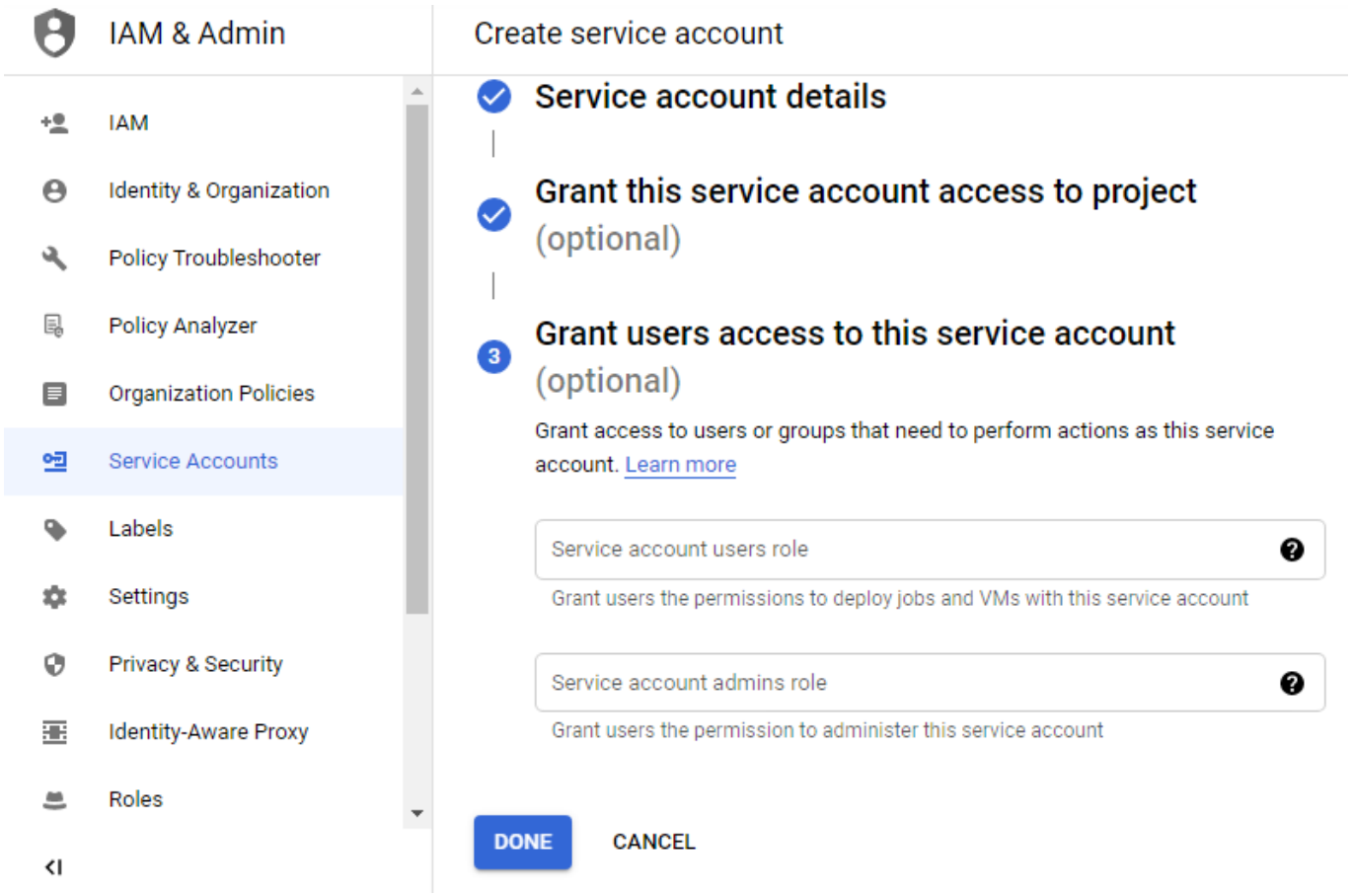
Select **CREATE SERVICE ACCOUNT** and a new page will be shown where you must set the service account name and description



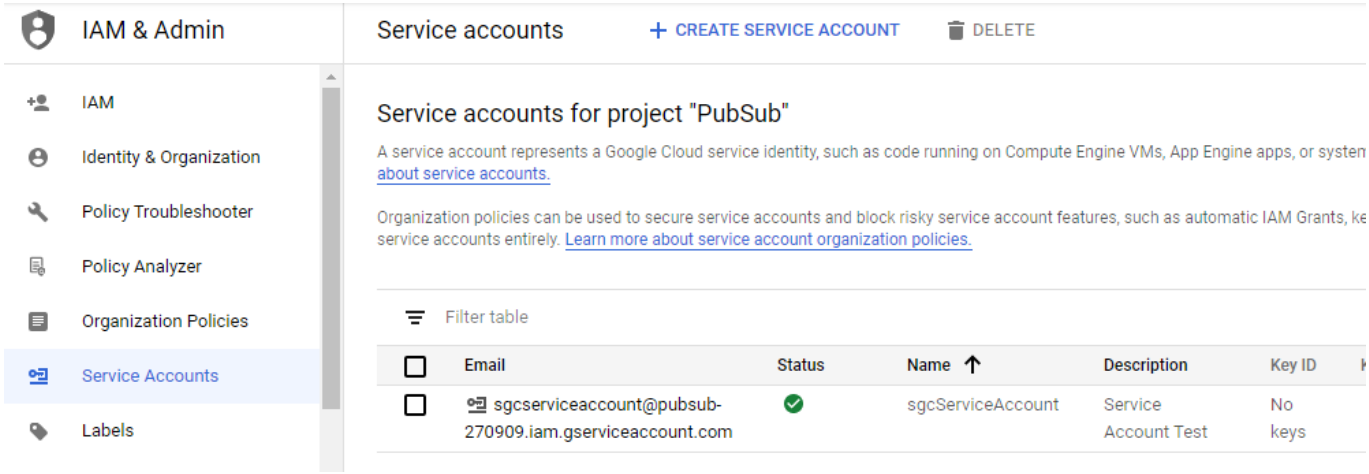
Then select at least one Role, I select PubSub Admin to allow the client publish and subscribe topics, but you can select other role with less privileges



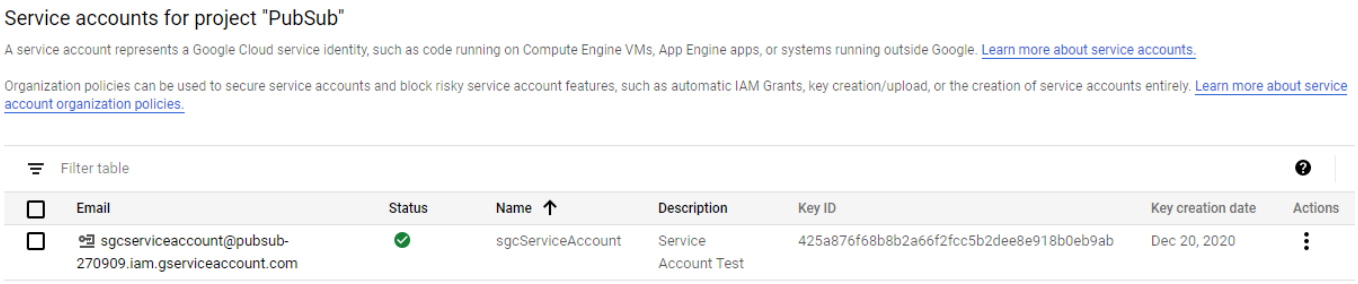
Press CONTINUE and finally you can grant access to other users



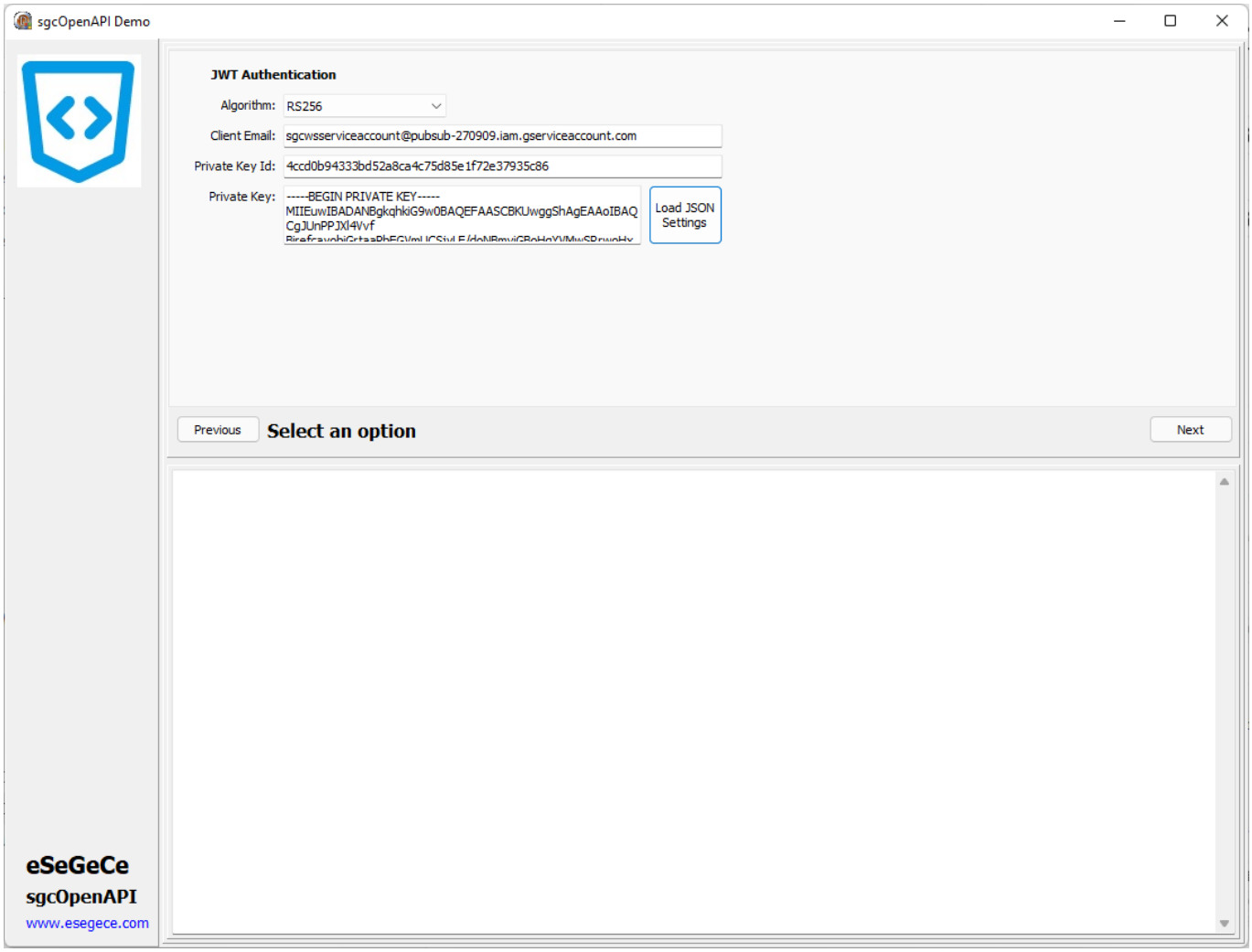
Press DONE when you finish and a new record will be shown



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/oauth
GetOpenAPIClient.Authentication.OAuth2.AuthorizationServerOptions.TokenURL := 'https://accounts.google.com/o/oaut
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/auth/
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/oauth
GetOpenAPIClient.Authentication.OAuth2.AuthorizationServerOptions.TokenURL := 'https://accounts.google.com/o/oaut
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/auth/
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/calendar';
GetOpenAPIClient.ListCalendarsEventsByCalendarId('email@mydomain.com', true, 'json', 0, '');
```

Insert Calendar Event

```
GetOpenAPIClient.InsertCalendarsEventsByCalendarId('email@mydomain.com', '{"summary":"Test Event","description":'
```

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.

```
procedure UploadFileToGoogleDrive;
var
  oDialog: TOpenDialog;
begin
  oDialog := TOpenDialog.Create(nil);
  Try
    if oDialog.Execute() then
    begin
      GetOpenAPIClient.OnUpload := OnUploadEvent;
      ShowMessage(GetOpenAPIClient.UploadFile(oDialog.FileName));
    end;
  Finally
    oDialog.Free;
  End;
end;
procedure OnUploadEvent(Sender: TObject; aState: TsgcOpenAPIProgressState; aValue: Int64);
begin
  case aState of
    oahpStart:
      begin
        ProgressBar1.Visible := True;
        ProgressBar1.Position := 0;
        ProgressBar1.Max := aValue;
      end;
    oahpProgress:
      begin
        ProgressBar1.Visible := True;
        ProgressBar1.Position := aValue;
      end;
    oahpEnd:
      begin
        ProgressBar1.Visible := False;
        ProgressBar1.Position := 0;
        ProgressBar1.Max := 0;
      end;
  end;
  Application.ProcessMessages;
end;
```

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.

```
procedure DownloadFileFromGoogleDrive;
var
  oSaveDialog: TSaveDialog;
begin
  oSaveDialog := TSaveDialog.Create(nil);
  Try
    if oSaveDialog.Execute then
    begin
      GetOpenAPIClient.OnDownload := OnDownloadEvent;
      GetOpenAPIClient.DownloadFile(txtDownloadFileId.Text, oSaveDialog.FileName);
    end;
  Finally
    oSaveDialog.Free;
  End;
```

```
end;
procedure OnDownloadEvent(Sender: TObject; aState: TsgcOpenAPIProgressState; aValue: Int64);
begin
  case aState of
    oahpStart:
      begin
        ProgressBar1.Visible := True;
        ProgressBar1.Position := 0;
        ProgressBar1.Max := aValue;
      end;
    oahpProgress:
      begin
        ProgressBar1.Visible := True;
        ProgressBar1.Position := aValue;
      end;
    oahpEnd:
      begin
        ProgressBar1.Visible := False;
        ProgressBar1.Position := 0;
        ProgressBar1.Max := 0;
      end;
  end;
  Application.ProcessMessages;
end;
```

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:

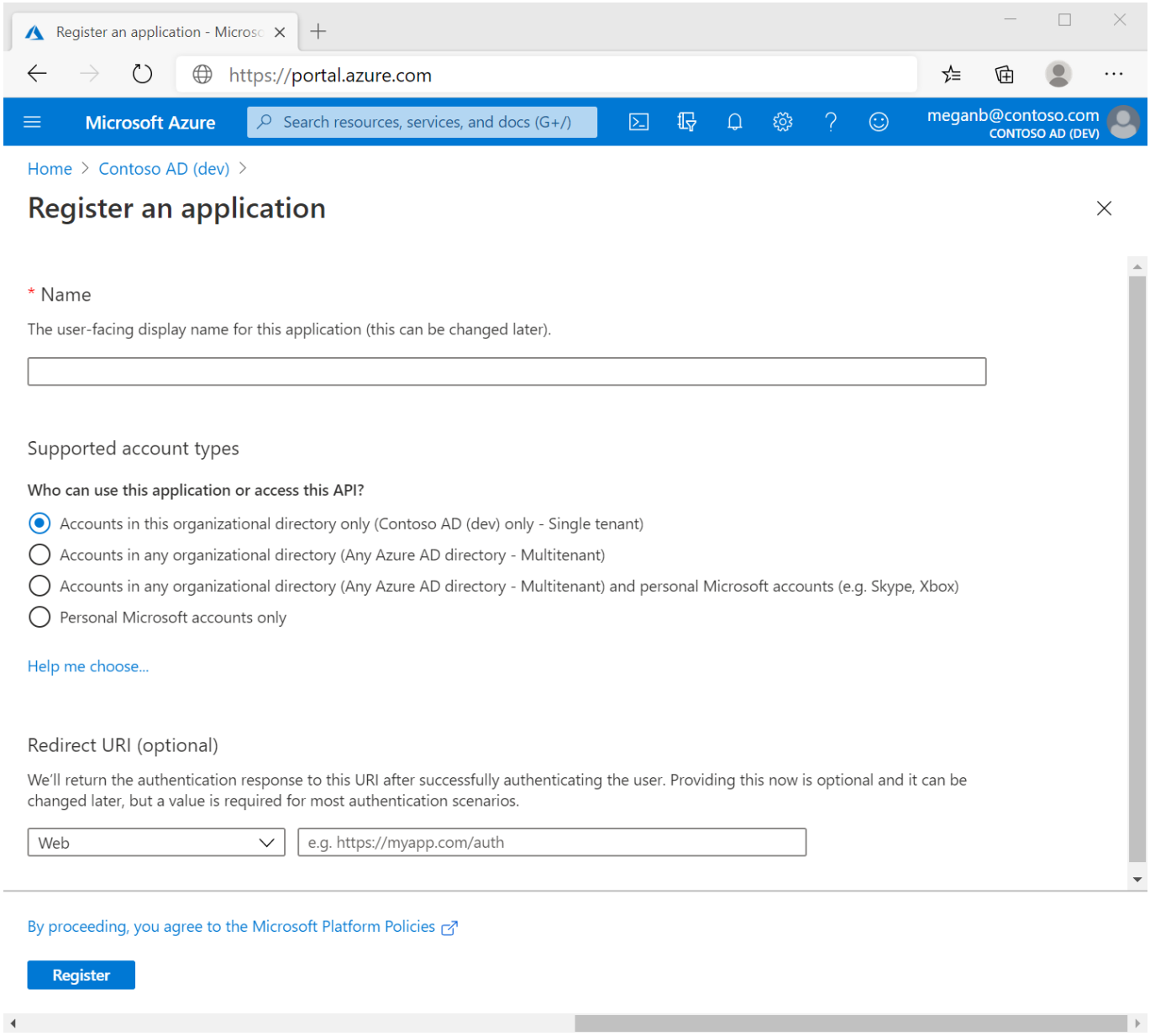
- **Organization name**
- **Initial domain** - Initial domain `<domainname>.onmicrosoft.com` can't be edited or deleted. You can add a customized domain name later.
- **Country or region**

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.

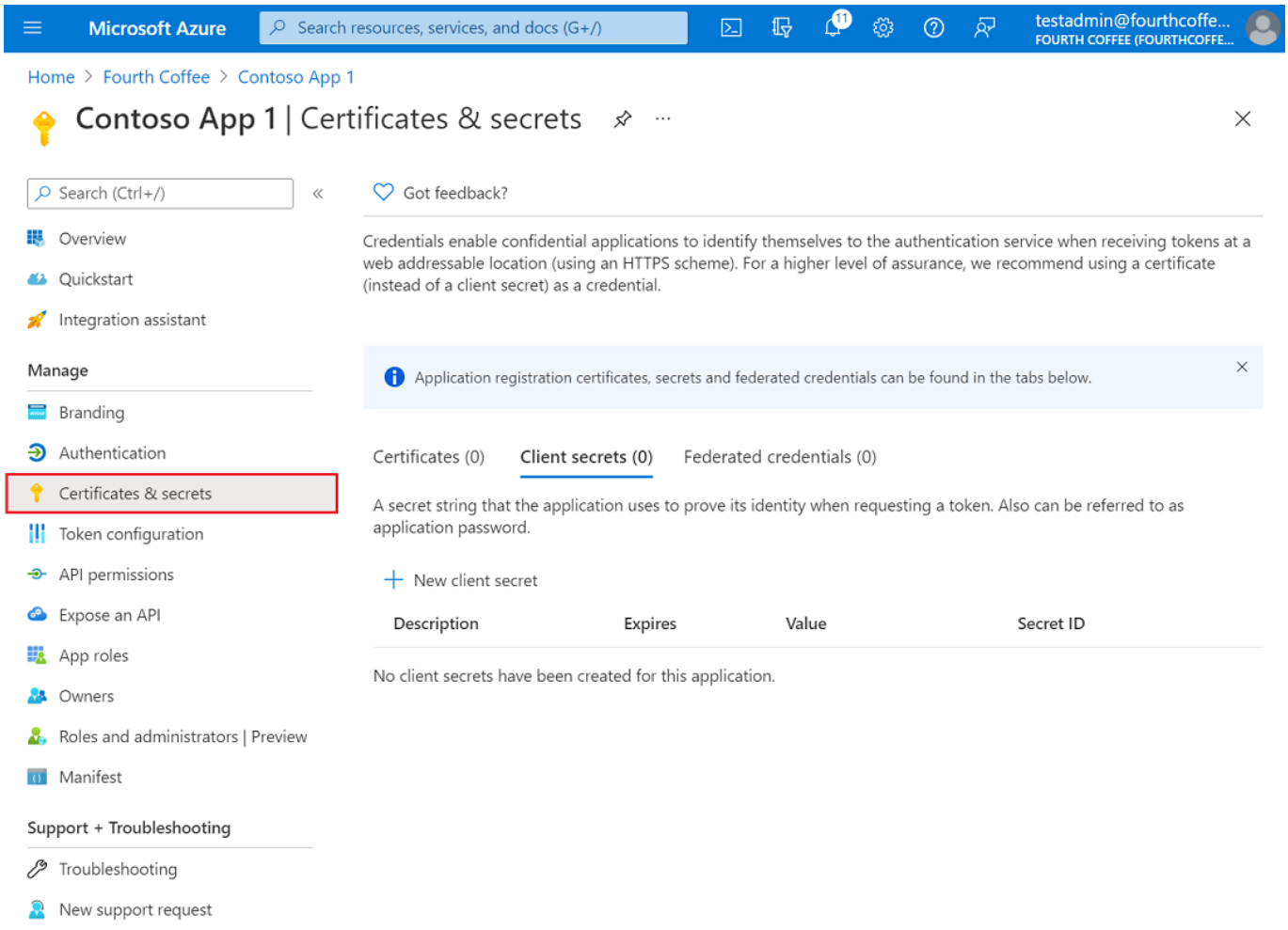
In a production web application, for example, the redirect URI is often a public endpoint where your app is running, like `https://contoso.com/auth-response`. During development, it's common to also add the endpoint where you run your app locally, like `https://127.0.0.1/auth-response` or `http://localhost/auth-response`.

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

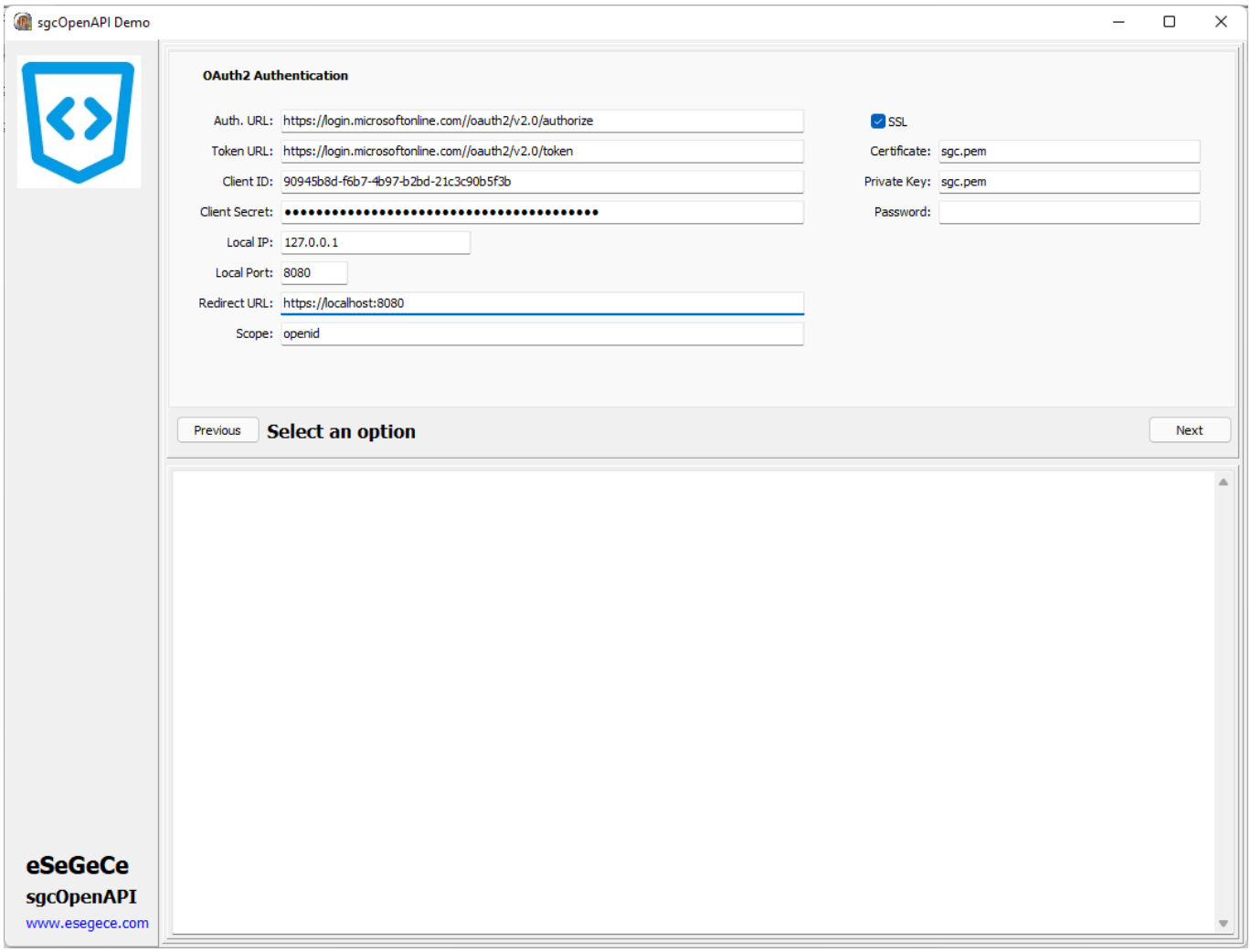
Using OAuth2 Code Grant Flow requires interaction with the user to login and get the required privileges.

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.com/t
GetOpenAPIClient.Authentication.OAuth2.AuthorizationServerOptions.TokenURL := 'https://login.microsoftonline.com/
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.

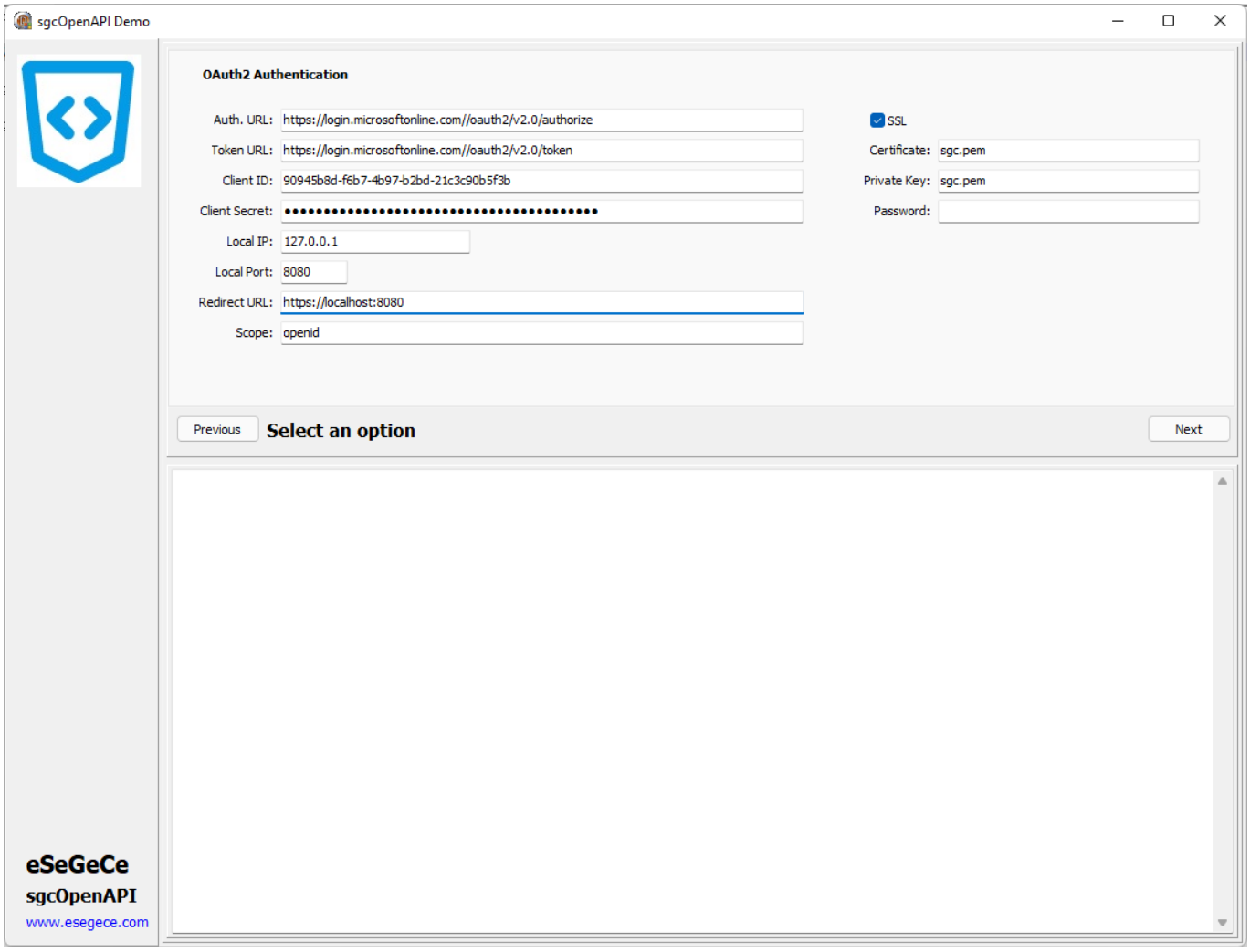
OpenAPI Microsoft | OAuth2 Credentials

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.com/t
GetOpenAPIClient.Authentication.OAuth2.AuthorizationServerOptions.TokenURL := 'https://login.microsoftonline.com/
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.com/t
GetOpenAPIClient.Authentication.OAuth2.AuthorizationServerOptions.TokenURL := 'https://login.microsoftonline.com/
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'));
```

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)

```
  WSServer.HTTP2Options.Enabled := chkHTTP2.Checked;
  if WSServer.HTTP2Options.Enabled then
  begin
    WSServer.SSLOptions.OpenSSL_Options.APIVersion := oslAPI_1_1;
    WSServer.SSLOptions.Version := tls1_3;
  end
  else
  begin
    case cboOpenSSLAPI.ItemIndex of
      0: WSServer.SSLOptions.OpenSSL_Options.APIVersion := oslAPI_1_0;
      1: WSServer.SSLOptions.OpenSSL_Options.APIVersion := oslAPI_1_1;
    end;
    case cboTLSVersion.ItemIndex of
      0: WSServer.SSLOptions.Version := tlsUndefined;
      1: WSServer.SSLOptions.Version := tls1_0;
      2: WSServer.SSLOptions.Version := tls1_1;
      3: WSServer.SSLOptions.Version := tls1_2;
      4: WSServer.SSLOptions.Version := tls1_3;
    end;
  end;
```

- 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
begin
  Port := StrToInt(txtDefaultPort.Text);
  IP := txtHost.Text;
end;
```

- 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

```
procedure TfrmServerChat.WSServerMessage(Connection: TsgcWSConnection; const Text: string);
begin
  WSServer.Broadcast(Text);
end;
```

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'

```
procedure TfrmServerChat.WSServerCommandGet(AContext: TIdContext; ARequestInfo:
    TIdHTTPRequestInfo; AResponseInfo: TIdHTTPResponseInfo);
begin
  if ARequestInfo.Document = '/jquery.js' then
  begin
    AResponseInfo.ContentText := pageJQuery.Content;
    AResponseInfo.ContentType := 'text/javascript';
    AResponseInfo.ResponseNo := 200;
  end;
end;
```

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 then
    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;
  end;
  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;
  end;
  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.

```
procedure TfrmClientChat.WSClientMessage(Connection: TsgcWSConnection; const
    Text: string);
begin
  memoLog.Lines.Add(Text);
end;
```

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...

```
procedure TfrmWebSocketClient.WSClientConnect(Connection: TsgcWSConnection);
begin
  DoLog('#connected');
end;
procedure TfrmWebSocketClient.WSClientDisconnect(Connection: TsgcWSConnection;
  Code: Integer);
begin
  DoLog('#disconnected');
end;
procedure TfrmWebSocketClient.WSClientError(Connection: TsgcWSConnection;
  const Error: string);
begin
  DoLog('#error: ' + Error);
end;
procedure TfrmWebSocketClient.WSClientException(Connection: TsgcWSConnection;
  E: Exception);
begin
  DoLog('#exception: ' + E.Message);
end;
procedure TfrmWebSocketClient.WSClientMessage(Connection: TsgcWSConnection;
  const Text: string);
begin
  DoLog(Text);
end;
```

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.

```
MQTT.Client := WSClient;
txtParameters.Text := '/';
chkTLS.Checked := False;
MQTT.Authentication.Enabled := False;
MQTT.Authentication.Username := '';
MQTT.Authentication.Password := '';
MQTT.MQTTVersion := mqtt311;
case aItemIndex of
  0: // esegece.com
    begin
      txtHost.Text := 'www.esegece.com';
      txtPort.Text := '15675';
      txtParameters.Text := '/ws';
      MQTT.Authentication.Enabled := True;
      MQTT.Authentication.Username := 'sgc';
      MQTT.Authentication.Password := 'sgc';
    end;
  1: // test.mosquitto.org
    begin
      txtHost.Text := 'test.mosquitto.org';
      txtPort.Text := '1883';
      chkTLS.Checked := False;
      chkOverWebSocket.Checked := False;
    end;
  2: // mqtt.fluux.io
    begin
      txtHost.Text := 'mqtt.fluux.io';
      txtPort.Text := '1883';
      chkTLS.Checked := False;
      chkOverWebSocket.Checked := False;
      MQTT.MQTTVersion := mqtt5;
    end;
  3: // broker.hivemq.com
    begin
      txtHost.Text := 'broker.mqttdashboard.com';
      txtPort.Text := '8000';
      txtParameters.Text := '/mqtt';
      chkTLS.Checked := False;
      chkOverWebSocket.Checked := True;
      MQTT.MQTTVersion := mqtt5;
    end;
end;
```

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...

```
procedure TfrmWebSocketClient.MQTTMQTTConnect(Connection: TsgcWSConnection;
    const Session: Boolean; const ReasonCode: Integer; const ReasonName:
    string; const ConnectProperties: TsgcWSMQTTCONNACKProperties);
begin
  DoLog('#connected');
  chkTLS.Enabled := False;
  chkCompressed.Enabled := False;
  if FMQTTSubscribeTopic  '' then
```

```
  begin
    MQTT.Subscribe(FMQTTSubscribeTopic);
    FMQTTSubscribeTopic := '';
  end;
end;



procedure TfrmWebSocketClient.MQTTMQTTPublish(Connection: TsgcWSConnection;
    aTopic, aText: string; PublishProperties: TsgcWSMQTTPublishProperties);
begin
  DoLog(aTopic + ': ' + aText);
end;



procedure TfrmWebSocketClient.MQTTMQTTSubscribe(Connection: TsgcWSConnection;
    aPacketIdentifier: Word; aCodes: TsgcWSSUBACKS; SubscribeProperties:
    TsgcWSMQTTSUBACKProperties);
begin
  DoLog('#Subscribe: ' + IntToStr(aPacketIdentifier));
end;



procedure TfrmWebSocketClient.MQTTMQTTDisconnect(Connection: TsgcWSConnection;
    ReasonCode: Integer; const ReasonName: string; DisconnectProperties:
    TsgcWSMQTTDISCONNECTProperties);
begin
  DoLog('#disconnected');
  chkTLS.Enabled := True;
  chkCompressed.Enabled := True;
end;
```

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.

```
SOCKETIO.Client := WSClient;
WSClient.Host := 'socketio-chat-h9jt.herokuapp.com';
WSClient.Port := 443;
WSClient.Options.Parameters := '/';
WSClient.Specifications.RFC6455 := True;
WSClient.TLS := 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

```
WSClient.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.

```
procedure TfrmWebSocketClient.WSClientMessage(Connection: TsgcWSConnection;
  const Text: string);
begin
  DoLog(Text);
end;
```

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

```
procedure TfrmServerSnapshots.DoBroadcastStream;
var
  oBitmap: TBitmap;
  oStream: TMemoryStream;
begin
  oBitmap := TBitmap.Create;
  try
    DoGetSnapshot(oBitmap);
    if WSServer.Active then
    begin
      oStream := TMemoryStream.Create;
      Try
        oBitmap.SaveToStream(oStream);
        oStream.Seek(0, soFromBeginning);
        WSServer.Broadcast(oStream);
      Finally
        FreeAndNil(oStream);
      End;
    end;
  finally
    oBitmap.FreeImage;
    FreeAndNil(oBitmap);
  end;
end;
```

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.

```
procedure TfrmClientSnapshots.WSClientBinary(Connection: TsgcWSConnection; const Data: TMemoryStream);
var
  oBitmap: TBitmap;
begin
  oBitmap := TBitmap.Create;
  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);
  End;
end;
```

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.

```
procedure TfrmServer.WSServerBinary(Connection: TsgcWSConnection; const Data: TMemoryStream);
var
  oFile: TFileStream;
begin
  if FFileName = '' then
    FFileName := FormatDateTime('yyyymmddhhnnsszz', Now) + '.dat';
  oFile := TFileStream.Create(FFileName, fmCreate);
  Try
    oFile.CopyFrom(Data, Data.Size);
  Finally
    oFile.Free;
  End;
  memoLog.Lines.Add('Received File: ' + FFileName);
end;
```

- 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

```
procedure TfrmServer.WSServerMessage(Connection: TsgcWSConnection; const Text: string);
begin
  if LeftStr(Text, Length(CS_UPLOAD_FILE)) = CS_UPLOAD_FILE then
    FFileName := MidStr(Text, Length(CS_UPLOAD_FILE) + 1, Length(Text))
  else
    memoLog.Lines.Add('Message Received (' + Connection.Guid + '): ' + Text);
end;
```

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.

```
procedure TfrmServer.WSServerAuthentication(Connection: TsgcWSConnection;
    aUser, aPassword: string; var Authenticated: Boolean);
begin
  if (aUser = 'user') and (aPassword = '1234') then
    Authenticated := True;
end;
```

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.

```
procedure TfrmServerPROTOCOL.WSServerCommandGet(AContext: TIdContext; ARequestInfo:
    TIdHTTPRequestInfo; AResponseInfo: TIdHTTPResponseInfo);
begin
  if ARequestInfo.Document = '/jquery.mobile.css' then
  begin
    AResponseInfo.ContentText := pageJQueryMobileCSS.Content;
    AResponseInfo.ContentType := 'text/css';
    AResponseInfo.ResponseNo := 200;
  end
  else if ARequestInfo.Document = '/jquery.js' then
  begin
    AResponseInfo.ContentText := pageJQuery.Content;
    AResponseInfo.ContentType := 'text/javascript';
    AResponseInfo.ResponseNo := 200;
  end
  else if ARequestInfo.Document = '/jquery.mobile.js' then
  begin
    AResponseInfo.ContentText := pageJQueryMobile.Content;
    AResponseInfo.ContentType := 'text/javascript';
    AResponseInfo.ResponseNo := 200;
  end
  else
  begin
    if AContext.Connection.Socket.Binding.Port = WSServer.SSLOptions.Port then
      FRequestSSL := True
    else
      FRequestSSL := False;
    AResponseInfo.ContentText := pageKendoUI_Grid.Content;
    AResponseInfo.ContentType := 'text/html';
    AResponseInfo.ResponseNo := 200;
  end;
end;
```

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.

```
procedure TfrmServerPROTOCOL.WSServerMessage(Connection: TsgcWSConnection; const Text: string);
var
  i: Integer;
  oJSON: TsgcJSON;
  oArray: IsgcObjectJSON;
  vText: string;
begin
  memoLog.Lines.Add(Connection.IP + ':' + Text);
  oJSON := TsgcJSON.Create(nil);
  Try
    oJSON.Read(Text);
    // ... read
    if oJSON.Node['type'].Value = 'read' then
    begin
      oArray := oJSON.AddArray('data');
      for i := 1 to 20 do
```

```
      begin
        With oArray.JSONObject.AddObject(IntToStr(i)).JSONObject do
        begin
          AddPair('ContactID', i);
          AddPair('ContactName', ContactName[i]);
          AddPair('ContactTitle', ContactTitle[i]);
          AddPair('CompanyName', CompanyName[i]);
          AddPair('Country', Country[i]);
        end;
      end;
      Connection.WriteData(oJSON.Text);
    end
    // ... update
    else if oJSON.Node['type'].Value = 'update' then
    begin
      WSServer.Broadcast(StringReplace(Text, '"type":"update"', '"type":"push-update"', []), '', '',
        Connection.Guid);
      Connection.WriteData(Text);
    end
    // ... destroy
    else if oJSON.Node['type'].Value = 'destroy' then
    begin
      WSServer.Broadcast(StringReplace(Text, '"type":"destroy"', '"type":"push-destroy"', []), '', '',
        Connection.Guid);
      Connection.WriteData(Text);
    end
    // ... create
    else if oJSON.Node['type'].Value = 'create' then
    begin
      vText := StringReplace(Text, 'null', formatDateTime('yyyymmddhhnnsszzz', Now), []);
      WSServer.Broadcast(StringReplace(vText, '"type":"create"', '"type":"push-create"', []), '', '',
        Connection.Guid);
      Connection.WriteData(vText);
    end;
  Finally
    FreeAndNil(oJSON);
  End;
end;
```

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.

```
procedure TfrmServer.Timer1Timer(Sender: TObject);
begin
  WSServer.Broadcast('data: ' + 'Server Time: ' + FormatDateTime('hh:nn:ss', Now));
end;
```

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
begin
  Port := StrToInt(txtDefaultPort.Text);
  IP := txtHost.Text;
end;
// ... 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 | Server AppRTC

This demo shows how to build a Video Conference Server using TsgcWebSocketHTTPServer and AppRTC 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_AppRTC.
- After that, you must assign the AppRTC Protocol to WebSocket Server and configure the server host and port. WebRTC requires secure connections, so you will need to use a PEM certificate and configure the SSLOptions property of the component.

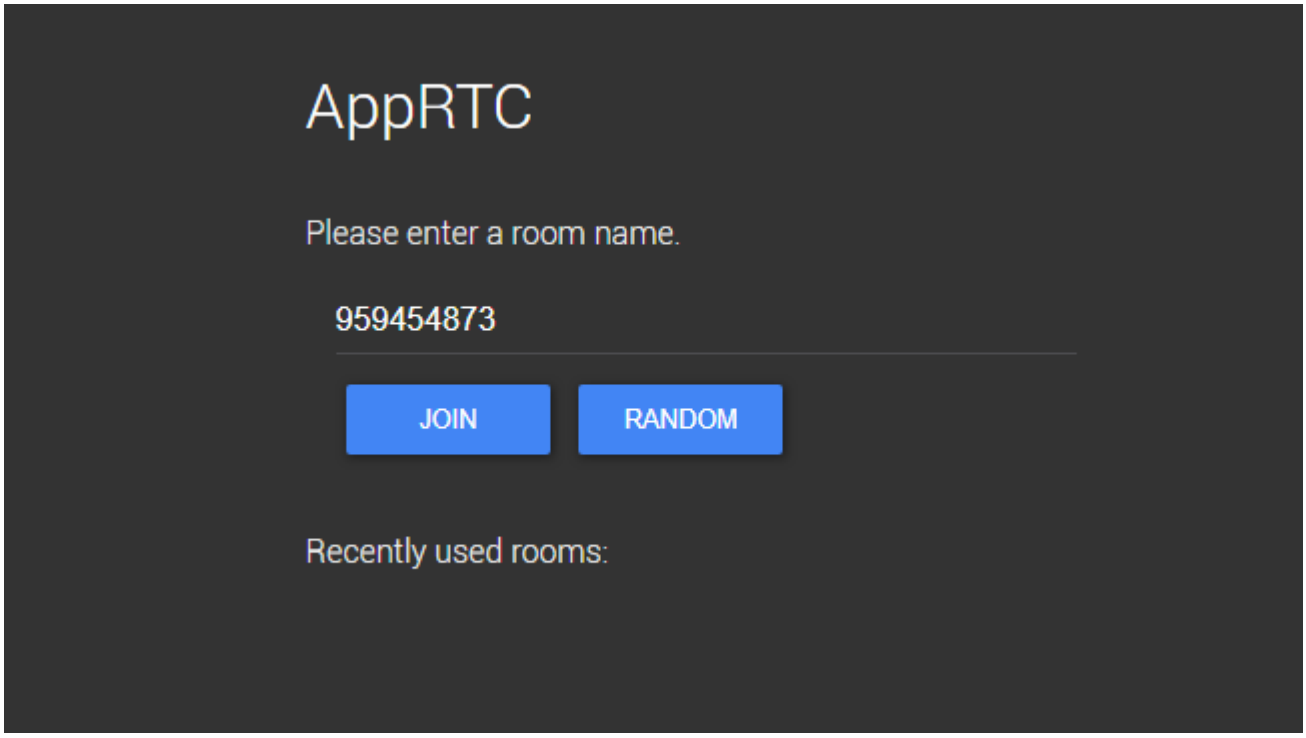
```
WSServer.Port := StrToInt(txtDefaultPort.Text);
// ... bindings
With WSServer.Bindings.Add do
begin
  Port := StrToInt(txtDefaultPort.Text);
  IP := txtHost.Text;
end;
// ... properties
WSPAppRTC.AppRTC.RoomLink := 'https://' + txtHost.Text + ':' + txtDefaultPort.Text + '/r/';
WSPAppRTC.AppRTC.WebSocketURL := 'wss://' + txtHost.Text + ':' + txtDefaultPort.Text;
// ... active
WSServer.Active := True;
```

- AppRTC.RoomLink is the url where the web-browser will be redirected to login to a room
- AppRTC.WebSocketURL is the url of the websocket connection
- The IceServers can be configured in the AppRTC Server protocol.

The demo requires an index HTML page which is used to dispatch the AppRTC 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 join a new room. Join this room and if successful you will see a link which must be used from another web-browser to start a new video-conference.


AppRTC
Please enter a room name.
959454873
JOIN
RANDOM
Recently used rooms:

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.

```
sgcTelegram.Telegram.API.ApiHash := txtApiHash.Text;
sgcTelegram.Telegram.API.ApiId := txtApiId.Text;
sgcTelegram.Telegram.PhoneNumber := '';
sgcTelegram.Telegram.BotToken := '';
if chkLoginBot.Checked then
  sgcTelegram.Telegram.BotToken := txtBotToken.Text
else
  sgcTelegram.Telegram.PhoneNumber := txtPhoneNumber.Text;


sgcTelegram.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.

```
procedure TFRMSGCTelegram.sgcTelegramAuthenticationCode(Sender: TObject;
  var Code: string);
begin
  Code := InputBox('Telegram', 'Introduce Telegram Code', '');
end;
```

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...

```
procedure TFRMSGCTelegram.sgcTelegramMessageText(Sender: TObject;
  MessageText: TsgcTelegramMessageText);
begin
  DoLogMessage(MessageText.ChatId, IntToStr(MessageText.SenderUserId),
    MessageText.Text);
end;
```

```
procedure TFRMSGCTelegram.sgcTelegramMessageDocument(Sender: TObject;
    MessageDocument: TsgcTelegramMessageDocument);
begin
  DoLogMessage(MessageDocument.ChatId, IntToStr(MessageDocument.SenderUserId),
    'New Document: ' + MessageDocument.FileName);
end;
```

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.

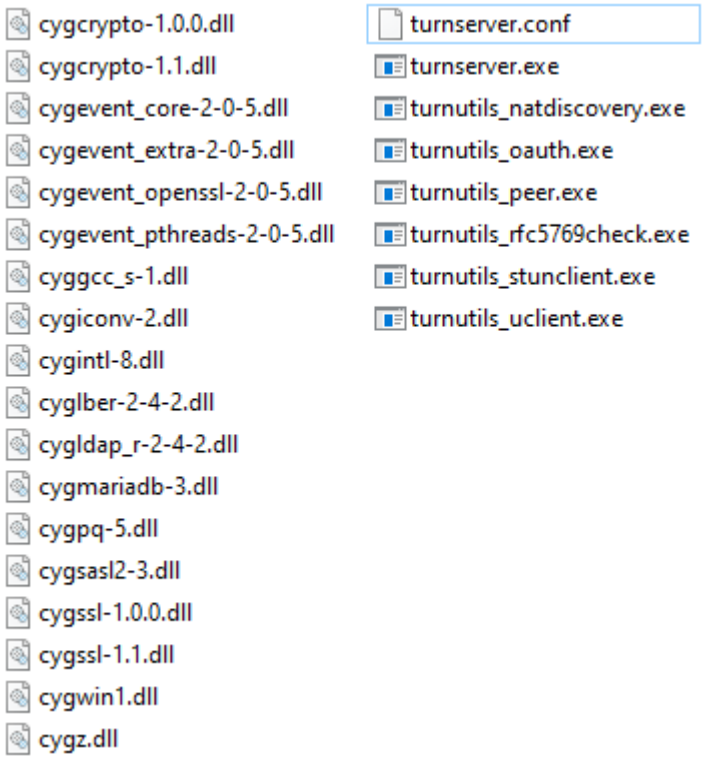
The supported project target platforms are:

- **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

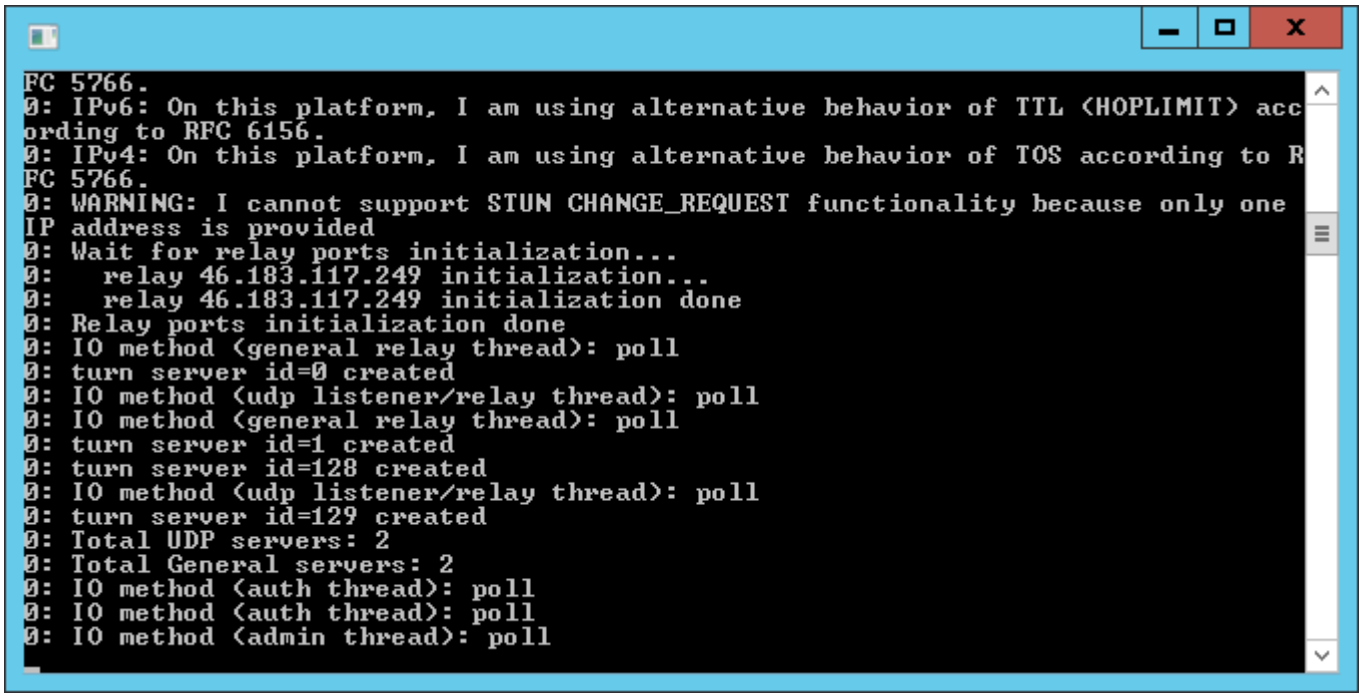
Server has the following public IP 80.15.44.123 and listens on port 80 and 443 (TLS connections). The credentials for connecting are: username = demo, password = secret. Your certificate name (must be in PEM format) is certificate.crt and private key is private.key.
Set the following configuration:

```
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. 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.

4. 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.
5. 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.
6. 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

AbstractApi Geolocation 1078
Add Telegram Proxy 646
ALPN 128
Amazon SQS 930
AMQP Channels 312
AMQP Consume Messages 320
AMQP Exchanges 314
AMQP Get Messages 322
AMQP Publish Messages 319
AMQP QoS 323
AMQP Queues 316
AMQP Transactions 324
AMQP1 Links 340
AMQP1 Read Message 347
AMQP1 Receiver Links 343
AMQP1 Send Message 345
AMQP1 Sender Links 341
AMQP1 Sessions 338
Anthropic 733
Anthropic Batches 745
Anthropic Citations 752
Anthropic Documents 748
Anthropic Extended Thinking 746
Anthropic Files 757
Anthropic MCP Connector 759
Anthropic Messages 738
Anthropic Models 744
Anthropic Prompt Caching 750
Anthropic Structured Outputs 755
Anthropic Tool Use 742
Anthropic Vision 740
Anthropic Web Search 753
API 3Commas 540
API Binance 410, 426, 432
API Binance Futures 426, 432
API Binance Futures Trade 432
API Bitfinex 510
API Bitget 595
API Bitmex 502
API Blockchain 569
API Bybit 554
API Cex 571
API Cex Plus 578
API Coinbase 435
API Coinbase Pro 435
API Crypto.com 602
API Deribit 599
API Discord 582
API Forex 559
API GateIO 597
API HTX 604
API Kraken 456, 458, 464, 467, 469, 472, 474, 481, 487, 489
API Kraken Futures 472, 474, 481, 487, 489
API Kraken Futures REST Private 489
API Kraken Futures REST Public 487
API Kraken REST Private 469
API Kraken REST Public 467
API Kucoin 514
API Kucoin Futures 528
API MEXC 587
API MEXC Futures 591
API OKX 545
API OpenAI 585
API Pusher 494
API SignalR 453
API SignalRCore 447
API SocketIO 433
API Telegram 630
API Whatsapp 606
API XTB 551
APIs 408, 410, 418, 421, 426, 432, 433, 435, 440, 443, 447, 453, 456, 458, 464, 467, 469, 472, 474, 481, 487, 489, 494, 502, 510, 540, 559, 566, 569, 571, 582, 587, 591, 630, 1077
APNs 830, 832, 833, 834
  Certificate-Based Connection 834
  Token-Based Connection 833
APP 830
Apple Push Notifications 829
Authentication 109, 166, 191
Authorization Code Grant 876

Authorization Code Grant (RFC 6749) 849
Authorization Code with PKCE (RFC 7636) 851
Auto Ban 238
Binance Connect 418
Binance Get Market Data 420
Binance Private Requests Time 424
Binance Private REST API 421
Binance Subscribe 419
Binance Trade Spot 422
Binance Withdraw 425
Binary Message 163
Bindings 118, 178
Bitmex Connect WebSocket API 506
Bitmex Subscribe WebSocket Channel 507
Bot 642
Broadcast 117
Brute Force 238
Build 72, 73, 75, 76
Build Android Application 75
Build iOS Application 76
Build OSX Application 73
Certificate-Based Connection 834
  APNs 834
Certificates OpenSSL 157
Certificates SChannel 158
Channels 117, 312, 419, 441, 567, 999
CIDR 236
Client 148, 150, 151, 161, 162, 163, 166, 169, 170, 171, 172, 196, 295, 297, 298, 310, 311, 395, 820, 821, 823, 824, 987, 988, 989, 996, 997, 998, 999, 1081, 1082, 1083, 1085, 1090, 1100
Client AMQP Connect 310
Client AMQP Disconnect 311
Client AMQP1 Authentication 335
Client AMQP1 Azure MessageBus 336
Client AMQP1 Connect 331
Client AMQP1 Connection State 334
Client AMQP1 Disconnect 332
Client AMQP1 Idle Timeout Connection 333
Client Authentication 166, 824
Client Chat 1081
Client Close Connection 150, 820
Client Connections 190
Client Credentials Grant 878
Client Credentials Grant (RFC 6749) 853
Client Exceptions 169
Client Keep Connection Active 821
Client Keep Connection Open 151
Client MQTT Connect 295
Client MQTT Sessions 297
Client MQTT Version 298
Client Open Connection 148
Client Pending Requests 823
Client Proxies 172
Client Register Protocol 171
Client Send Binary Message 162
Client Send Text 161, 163
Client Send Text Message 161
Client Snapshots 1090
Client SocketIO 1085
Client WebSocket HandShake 170
Clients 297, 310, 395, 821, 823, 1081, 1083
  Send Files 395
Coinbase Pro Connect 440
Coinbase Pro Get Market Data 442
Coinbase Pro Place Orders 445
Coinbase Pro Private Requests Time 444
Coinbase Pro Private REST API 443
Coinbase Pro SandBox Account 446
Coinbase Pro Subscribe 441
Compression 121
Configure Install 55
Configure ZLib 63
Connect Mosquitto 296
Connect Secure Server 156
Connect TCP Server 153
Connect WebSocket Server 147
Connections TIME_WAIT 154
Coturn 1102
Cross-Site Scripting 240
CryptoHopper 649
Custom Objects 123
Custom Sub 106
Datasnap 1016
DeepSeek 764
DeepSeek Messages 766
DeepSeek Models 768
DeepSeek Vision 767
Deflate-Frame 661

Device Authorization Grant (RFC 8628) 857, 884
DPoP - Demonstrating Proof of Possession (RFC 9449) 859
DPoP Validation (RFC 9449) 890
Dropped Disconnections 152
Editions 28
Embeddings ChatBot 729
Embeddings Create Vectors 728
EPOLL 127
Error 207
Extensions 659
Fast Performance Server 78
Files 119, 142, 200, 394, 395, 396, 817, 1091
Fired 227
Firewall Blacklist 236
Firewall Whitelist 236
Flash 122
Flood Protection 243
Flow 70
  Threading 70
Forex Account 564
Forex Connect 561, 566
Forex Market Data 565
Forex Subscribe 567
Forex Trade 562
Forward HTTP Requests 129
Found 642
Fragmented Messages 144
Gemini 761
Gemini CountTokens 776
Gemini EmbedContent 777
Gemini Embeddings 777
Gemini Function Calling 778
Gemini Messages 771
Gemini Models 774
Gemini Structured Outputs 775
Gemini Token Counting 776
Gemini Tool Use 778
Gemini Vision 773
Generate 831
  Remote Notification APNs 831
Google Calendar 954, 960, 961, 962
Google Calendar RefreshToken 962
Google Calendar Service Account 963
Google Calendar Sync Calendars 960
Google Calendar Sync Events 961
Google Cloud FCM 964
Google Cloud Pub 945
Google OAuth2 Keys 935
Google Service Accounts 941
Grok 786
Grok Messages 788
Grok Models 790
Grok Vision 789
Groups 124
HeartBeat 113
HighPerfAcceptsPerWorker 218
History 564
How to Place a Bitmex Order 508
HTTP 68, 116, 129, 199, 200, 207, 208, 222, 811, 815
HTTP Dispatch Files 200
HTTP Server Stream Video 210
HTTP/2 201, 202, 204, 816, 817, 818, 819, 822, 825
HTTP/2 Alternate Service 204
HTTP/2 Download File 817
HTTP/2 Headers 819
HTTP/2 Partial Responses 818
HTTP/2 Reason Disconnection 822
HTTP/2 Server Push 202, 816
HTTP/2 Server Threads 205
HTTP1 836
HTTP2 812
HTTPAPI 220, 222, 223, 227
HTTPAPI Custom Headers 223
HTTPAPI Disable HTTP 222
HTTPAPI FineTune 218
HTTPAPI OnDisconnect 227
HTTPAPI Send File Response 225
HTTPAPI Send Text Response 224
HTTPAPI URL Reservation 216
ICE 1003
ICE Gather Candidates 1005
ICE Pair Candidates 1006
In HTML 870
Indy 99
Indy SChannel 183, 211
Install Errors 51
Install Package 44
Install Setup 33

Install sgcIndy Package 57
Installation 30
Introduction 23
IOCP 126
IoT 798, 799, 806
IoT Amazon MQTT Client 799
IoT Azure MQTT Client 806
IP Ban 238
IP Filtering 236
JWT 899
KendoUI_Grid 1094
Kucoin Connect WebSocket API 521
Kucoin Futures Connect WebSocket API 533
Kucoin Futures Get Market Data 535
Kucoin Futures Private Requests Time 539
Kucoin Futures Private REST API 536
Kucoin Futures Subscribe WebSocket Channel 534
Kucoin Futures Trade 537
Kucoin Get Market Data 523
Kucoin Private Requests Time 527
Kucoin Private REST API 524
Kucoin Subscribe WebSocket Channel 522
Kucoin Trade Spot 525
License 1104
Lightstreamer 566, 567
LoadBalancing 141
Logs 115
MaxConnectionsPerIP 243
MaxMessagesPerSec 243
MCP 662
MCP Client Elicitation 687
MCP Client Prompts 681
MCP Client Resources 683
MCP Client Roots 685
MCP Client Sampling 686
MCP Client Tools 679
MCP Server Elicitation 676
MCP Server Prompts 669
MCP Server Resources 672
MCP Server Roots 674
MCP Server Sampling 675
MCP Server Sessions 665
MCP Server Tools 666
Memory Manager 81
Message Filtering 240
Method 815
Mistral AI 791
Mistral Embeddings 797
Mistral Messages 794
Mistral Models 796
Mistral Vision 795
MQTT 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 799, 806, 1083
MQTT Clear Retained Messages 305
MQTT Publish 299, 302, 304
MQTT Publish Message 302
MQTT Publish Subscribe 299
MQTT Receive Messages 303
MQTT Subscribe 301
MQTT Topics 300
Notification Requests 832
  Sending 832
OAuth2 825, 840, 866, 870, 871, 872, 873, 874, 875, 935
OAuth2 Client for Desktop Applications 846
OAuth2 Client for Web Applications 845
OAuth2 Customize Sign 870
OAuth2 None Authenticate URLs 875
OAuth2 Provider Authentication 896
OAuth2 Provider Azure AD 894
OAuth2 Provider Private Endpoints 895
OAuth2 Provider Requests 898
OAuth2 Recover Access Tokens 873
OAuth2 Register Apps 872
Ollama 780
Ollama Embeddings 785
Ollama Messages 783
Ollama Models 784
ompClassic 218
ompHighPerf 218
OpenAI 688
OpenAI Applications 707
OpenAI Audio 699, 716
OpenAI Batch 705
OpenAI Chat 697
OpenAI Completion 696
OpenAI Edit 698
OpenAI Fine-Tuning 704
OpenAI Moderation 700
OpenAI RealTime 701
OpenAI Responses 702

OpenAI Speech 703
OpenAI Uploads 706
OpenAPI 1021
OpenAPI Additional Properties 1027
OpenAPI Amazon AWS 1037
OpenAPI Amazon AWS Credentials 1043
OpenAPI Amazon AWS S3 1045
OpenAPI API 1033
OpenAPI Client 1034
OpenAPI Command Line 1029
OpenAPI Google Cloud 1046
OpenAPI Google Cloud Calendar 1060
OpenAPI Google Cloud OAuth2 1051
OpenAPI Google Cloud PubSub 1059
OpenAPI Google Cloud Service Accounts 1054
OpenAPI Google Drive 1061
OpenAPI Library 1031
OpenAPI Microsoft 1063
OpenAPI Microsoft Graph 1076
OpenAPI Microsoft OAuth2 Code 1072
OpenAPI Microsoft OAuth2 Credentials 1074
OpenAPI Microsoft Register Application 1069
OpenAPI Microsoft Tenant 1068
OpenAPI Parser Pascal 1022
OpenSSL 84, 87, 89, 91, 92, 157
OpenSSL Android 91
OpenSSL iOS 92
OpenSSL Load Additional Functions 98
OpenSSL OSX 89
OpenSSL Own CA Certificates 94
OpenSSL P12 Certificates 96
OpenSSL Verify Certificate 97
OpenSSL Windows 87
OperatingMode 218
Orders 562, 564
Overview 64
PDF-Back-Cover 1114
PDF-Front-Cover 1
PerMessage-Deflate 660
Pinecone 730
Positions 564
Post Big Files 119
Price Bars 565
Price Ticks 565
Protocol AMQP 306
Protocol AMQP1 326
Protocol AppRTC 352
Protocol Dataset 381, 385, 388, 389
Protocol Dataset Javascript 385
Protocol Dataset Notify Updates 389
Protocol Dataset Replicate Table 388
Protocol Default 372, 377
Protocol Default Javascript 377
Protocol E2EE 404
Protocol Files 390
Protocol MQTT 292
Protocol Presence 397, 401
Protocol Presence Javascript 401
Protocol STOMP 348
Protocol WAMP 357, 360
Protocol WAMP Javascript 360
Protocol WAMP2 368
Protocol WebRTC 354, 356
Protocol WebRTC Javascript 356
Protocols 106, 171, 287, 289, 292, 306, 348, 352, 354, 356, 357, 360, 368, 372, 377, 381, 385, 388, 389, 390, 397, 401
Protocols Javascript 289
Proxy 143, 172, 646
Quality 130
  Service 130
Quality Of Service 130
QueueLength 218
Queues 132, 316
QuickStart HTTP 68
QuickStart WebSockets 66
Rate Limiting 243
RCON 648
Receive Binary Messages 165, 195
Receive Text Messages 164, 194
Refresh Token Grant 882
Register 647, 830
Register Telegram User 647
Remote Notification APNs 831
  Generate 831
Request HTTP 815
Resource Owner Password Credentials Grant 880
Resource Owner Password Credentials Grant (RFC 6749) 855

Response Body 207
REST API 561
RTCMultiConnection 654
RTCPeerConnection Data 1015
RTCPeerConnection DTLS 1014
RTCPeerConnection ICE 1013
RTCPeerConnection Signaling 1012
RTCPeerConnection STUN TURN 1011
RTCPeerConnection WebSocket Client 1010
RTCPeerConnection WebSocket Server 1009
SChannel 183, 211
SChannel Get Connection Info 160
SChannel Server 183, 211
Secure Connections 111
Self-Signed Certificates 221
Send Big Files 396
Send Files 394, 395
  Clients 395
  Server 394
Send Files To Clients 395
Send Files To Server 394
Send Telegram Invoice Message 644
Send Telegram Message Bold 641
Send Telegram Message With Buttons 639, 640
Send Telegram Message With Inline Buttons 639
Sending 832
  Notification Requests 832
Server 177, 178, 179, 180, 181, 187, 188, 189, 191, 192, 193, 194, 195, 196, 199, 201, 208, 220, 296, 394, 816, 866, 871, 874, 991, 992, 1001, 1002, 1079, 1086, 1089, 1093, 1097, 1098
  Send Files 394
Server AppRTC 1098
Server Authentication 191, 874, 1093
Server Bindings 178
Server Chat 1079
Server Close Connection 189
Server Endpoints 871
Server Example 866
Server Keep Active 180
Server Keep Connections Alive 187
Server Monitor 1086
Server Plain TCP 188
Server Read Headers 196
Server Requests 199
Server Send Binary Message 193
Server Send Text Message 192
Server Sessions 208
Server Snapshots 1089
Server SSL 181, 220
Server SSL SChannel 183, 211
Server Start 177
Server Startup Shutdown 179
Server Verify Certificate 186
Server-Sent Events 139
ServerSentEvents 1096
Service 130, 204, 941
  Quality 130
SkipIOCPOnSuccess 218
SQL Injection 240
Stop Limit 562
STUN 984, 987, 988, 989, 991, 992
STUN Client Attributes 989
STUN Client Long Term Credentials 988
STUN Client UDP Retransmissions 987
STUN Server Alternate Server 992
STUN Server Long Term Credentials 991
Sub 945
SubProtocol 137
TCP Connections 136
Telegram Chat 642
Telegram Client 1100
Telegram Get SuperGroup Members 645
Telegram Sponsored Messages 643
Threading 70
  Flow 70
Throttle 138
Token Introspection (RFC 7662) 888
Token Revocation (RFC 7009) 886
Token-Based Connection 833
  APNs 833
Transactions 134
TsgcAIChat 722
TsgcAIDatabaseVectorFile 726
TsgcAIDatabaseVectorPinecone 727
TsgcAIOpenAIAssistant 708
TsgcAIOpenAIAssistant File Search 710
TsgcAIOpenAIAssistant Function Calling 714
TsgcAIOpenAIAssistant Streaming 712

TsgcAIOpenAIChatBot 723
TsgcAIOpenAIEmbeddings 725
TsgcAIOpenAITranslator 724
TsgcAudioPlayerMCI 718
TsgcAudioRecorderMCI 717
TsgcHTTP_API_Anthropic 737
TsgcHTTP_API_DeepSeek 765
TsgcHTTP_API_Gemini 763
TsgcHTTP_API_Grok 787
TsgcHTTP_API_Mistral 793
TsgcHTTP_API_Ollama 782
TsgcHTTP_API_OpenAI 695
TsgcHTTP_JWT_Client 901
TsgcHTTP_JWT_Server 904
TsgcHTTP_OAuth2_Client 841
TsgcHTTP_OAuth2_Client_Google 847
TsgcHTTP_OAuth2_Client_Microsoft 848
TsgcHTTP_OAuth2_Server 863
TsgcHTTP_OAuth2_Server_Provider 892
TsgcHTTP2Client 813
TsgcHTTP2ConnectionClient 826
TsgcHTTP2RequestProperty 827
TsgcHTTP2ResponseProperty 828
TsgcICEClient 1004
TsgcIWWebSocketClient 283
TsgcIWWSPClient_Dataset 384
TsgcIWWSPClient_sgc 376
TsgcRTCPeerConnection 1007
TsgcServerHTTPAPI_FineTune 218
TsgcSTUNClient 985
TsgcSTUNServer 990
TsgcTCPClient 173
TsgcTextToSpeechAmazon 721
TsgcTextToSpeechGoogle 720
TsgcTextToSpeechSystem 719
TsgcTURNClient 994
TsgcTURNServer 1000
TsgcUDPCLient 982
TsgcUDPServer 983
TsgcWebPush_Client 658
TsgcWebSocketClient 145
TsgcWebSocketClient_WinHTTP 228
TsgcWebSocketFirewall 229, 236, 238, 240, 243
TsgcWebSocketHTTPServer 197
TsgcWebSocketHTTPServer_Sessions 208
TsgcWebSocketLoadBalancerServer 281
TsgcWebSocketProxyServer 282
TsgcWebSocketServer 175
TsgcWebSocketServer_HTTPAPI 214
TsgcWebView2 967
TsgcWSAPIClient_MCP 678
TsgcWSAPIKeyManager 269
TsgcWSAPIServer_MCP 663
TsgcWSAPIServer_WebAuthn 906
TsgcWSAPIServer_WebPush 657
TsgcWSCircuitBreaker 258
TsgcWSConnection 285
TsgcWSHTTP2WebBrokerBridgeServer 1019
TsgcWSHTTPWebBrokerBridgeServer 1017
TsgcWSMessageFile 393
TsgcWSPClient_AMQP 307
TsgcWSPClient_AMQP1 328
TsgcWSPClient_Dataset 383
TsgcWSPClient_E2EE 407
TsgcWSPClient_Files 392
TsgcWSPClient_MQTT 294
TsgcWSPClient_Presence 400
TsgcWSPClient_sgc 375
TsgcWSPClient_STOMP 349
TsgcWSPClient_STOMP_ActiveMQ 351
TsgcWSPClient_STOMP_RabbitMQ 350
TsgcWSPClient_WAMP 359
TsgcWSPClient_WAMP2 369
TsgcWSPresenceMessage 399
TsgcWSPServer_AppRTC 353
TsgcWSPServer_Dataset 382
TsgcWSPServer_E2EE 406
TsgcWSPServer_Files 391
TsgcWSPServer_Presence 398
TsgcWSPServer_sgc 374
TsgcWSPServer_WAMP 358
TsgcWSPServer_WebRTC 355
TsgcWSRateLimiter 246
TsgcWSServer_HTTPAPI_WebBrokerBridge 1020
TURN 993, 996, 997, 998, 999, 1001, 1002
TURN Client Allocate IP Address 996
TURN Client Create Permissions 997
TURN Client Send Indication 998
TURN Server Allocations 1002
TURN Server Long Term Credentials 1001

Upload File 1091
Using DLL 103
Wait Response 304
WAMP 363, 364, 365, 366
WAMP Publishers 364
WAMP RPC Progress Results 366
WAMP Simple RPC 365
WAMP Subscribers 363
WatchDog 114
Web Browser Test 105
WebAuthn 905
WebAuthn Authentication 917
WebAuthn Authentication Request 919
WebAuthn Authentication Response 920
WebAuthn Authentication Result 921
WebAuthn Authorization 923
WebAuthn Authorization HTTP 924
WebAuthn Authorization WebSocket 925
Webauthn Javascript Client 926
WebAuthn MDS 922
WebAuthn Registration 910
WebAuthn Registration Request 913
WebAuthn Registration Response 914
WebAuthn Registration Result 916
WebPush 656
WebRTC 1097
WebSocket Events 101
WebSocket Parameters Connection 102
WebSocket Redirections 155
WebSockets 66, 101, 102, 147, 155, 170, 418, 419, 440, 441, 458, 464, 474, 481, 566, 567
WebSockets Private 464, 481
WebSockets Public 458, 474
WebView2 Advanced Features 979
WebView2 Cookies 973
WebView2 Downloads 975
WebView2 JavaScript 971
WebView2 Navigation 969
WebView2 Settings 977
WhatsApp Create App 610
WhatsApp Download Media 629
WhatsApp Phone Number Id 612
WhatsApp Receive Messages and Status Notifications 625
WhatsApp Security 615
WhatsApp Send Files 627
WhatsApp Send Interactive Messages 619
WhatsApp Send Messages 616
WhatsApp Send Template Messages 623
WhatsApp Token 613
WhatsApp Webhook 614
Windows Service 104
XSS 240

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