



sgcWebSockets 2026.6

June 2026

CBuilder 文档

Copyright © 2012-2026 eSeGeCe Software

info@esegece.com

www.esegece.com

目录

简介	23
概述	27
版本	27
安装	29
安装	32
安装设置	32
安装软件包	43
安装错误	50
配置安装	53
安装 sgclndy 包	55
配置 ZLib	60
快速入门	61
概述	61
WebSockets 快速入门	63
HTTP 快速入门	65
线程流程	67
构建	68
构建 OSX 应用程序	69
构建 Android 应用程序	71
构建 iOS 应用程序	72
高性能服务器	74
内存管理器	77
OpenSSL	80
OpenSSL Windows	83
OpenSSL OSX	85

OpenSSL Android	87
OpenSSL iOS	88
OpenSSL 自有 CA 证书	90
OpenSSL P12 证书	92
OpenSSL 验证证书	93
OpenSSL 加载附加函数.....	94
Indy	96
主题	98
WebSocket 事件	98
WebSocket 参数连接	99
在DLL内使用	100
Windows 服务	101
Web 浏览器测试	102
自定义子协议	103
身份验证.....	104
安全连接.....	106
HeartBeat	108
WatchDog.....	109
日志	110
HTTP.....	111
广播与频道	112
绑定	113
上传大文件	114
压缩	116
Flash.....	117
自定义对象	118
群组	119
IOCP	121
EPOLL.....	122
ALPN	123

转发 HTTP 请求.....	124
服务质量.....	125
队列	127
事务	129
TCP 连接.....	130
SubProtocol	131
Throttle	132
服务器发送事件	133
LoadBalancing	135
文件	136
Proxy.....	137
分片消息.....	138
组件	139
TsgcWebSocketClient.....	139
连接 WebSocket 服务器.....	141
客户端打开连接	142
客户端关闭连接	144
客户端保持连接打开	145
丢失的断开连接	146
连接 TCP 服务器	147
连接 TIME_WAIT	148
WebSocket 重定向	149
连接安全服务器	150
OpenSSL 证书.....	151
SChannel 证书.....	152
SChannel 获取连接信息.....	153
客户端发送文本消息	154
客户端发送二进制消息	155
客户端发送文本和二进制消息	156
接收文本消息	157

接收二进制消息	158
客户端认证	159
客户端异常	162
Client WebSocket HandShake	163
Client Register Protocol	164
客户端代理	165
TsgcTCPClient	166
TsgcTCPClient	167
TsgcWebSocketServer	168
服务器启动	170
服务器绑定	171
服务器启动和关闭	172
服务器保持活跃	173
服务器 SSL	174
服务器 SSL SChannel	176
服务器验证证书	179
服务器保持连接存活	180
服务器纯 TCP	181
服务器关闭连接	182
客户端连接	183
服务器身份验证	184
服务器发送文本消息	185
服务器发送二进制消息	186
服务器接收文本消息	187
服务器接收二进制消息	188
服务器从客户端读取请求头	189
TsgcWebSocketHTTPServer	190
HTTP 服务器请求	192
HTTP 分发文件	193
HTTP/2 服务器	194
HTTP/2 服务器推送	195

HTTP/2 备用服务	197
HTTP/2 服务器线程	198
无响应主体的HTTP 404错误	200
HTTP 服务器会话	201
HTTP 服务器流式视频	203
服务器 SSL SChannel	204
TsgcWebSocketServer_HTTPAPI	207
HTTPAPI URL 保留	209
HTTPAPI FineTune	211
HTTPAPI 服务器 SSL	213
自签名证书	214
HTTPAPI 禁用 HTTP/2	215
HTTPAPI 自定义请求头	216
HTTPAPI 发送文本响应	217
HTTPAPI 发送文件响应	218
HTTPAPI OnDisconnect 未触发	220
TsgcWebSocketClient_WinHTTP	221
TsgcWebSocketFirewall	222
防火墙：黑名单与白名单	228
防火墙：暴力破解保护	230
防火墙：SQL 注入和 XSS 检测	232
防火墙：速率限制与洪水防护	235
防火墙：Bot 检测	237
TsgcWSRateLimiter	240
TsgcWSCircuitBreaker	251
TsgcWSAPIKeyManager	261
TsgcWebSocketLoadBalancerServer	272
TsgcWebSocketProxyServer	273
TsgcIWWebSocketClient	274
TsgcWSConnection	276
协议	278

协议 Javascript.....	280
协议 MQTT.....	283
TsgcWSPClient_MQTT	284
客户端 MQTT 连接.....	285
连接 Mosquitto MQTT 服务器.....	286
客户端 MQTT 会话.....	287
客户端 MQTT 版本.....	288
MQTT 发布订阅	289
MQTT 主题.....	290
MQTT 订阅.....	291
MQTT 发布消息	292
MQTT 接收消息	293
MQTT 发布并等待响应.....	294
MQTT 清除保留消息.....	295
Protocol AMQP	296
TsgcWSPClient_AMQP.....	297
Client AMQP Connect	300
客户端 AMQP 断开连接.....	301
AMQP 通道	302
AMQP 交换机.....	304
AMQP 队列	306
AMQP 发布消息.....	309
AMQP 消费消息.....	310
AMQP 获取消息.....	312
AMQP QoS.....	313
AMQP 事务	314
协议 Kafka.....	315
TsgcWSPClient_Kafka.....	316
Client Kafka Connect.....	318
客户端 Kafka 断开连接.....	319
Kafka 生产消息	320

Kafka 消费消息	321
Kafka 消费者组	322
Kafka 主题.....	323
Kafka 偏移量.....	324
Protocol AMQP1	325
TsgcWSPClient_AMQP1	326
Client AMQP1 Connect	329
客户端 AMQP1 断开连接	330
客户端 AMQP1 空闲超时连接.....	331
Client AMQP1 连接状态.....	332
Client AMQP1 身份验证.....	333
客户端 AMQP1 Azure MessageBus.....	334
AMQP1 会话	336
AMQP1 链路	338
AMQP1 发送方链路	339
AMQP1 接收方链路	341
AMQP1 发送消息.....	343
AMQP1 读取消息.....	345
Protocol STOMP	346
TsgcWSPClient_STOMP	347
TsgcWSPClient_STOMP_RabbitMQ.....	348
TsgcWSPClient_STOMP_ActiveMQ	349
协议 AppRTC.....	350
TsgcWSPServer_AppRTC.....	351
协议 WebRTC.....	352
TsgcWSPServer_WebRTC.....	353
协议 WebRTC JavaScript.....	354
协议 WAMP.....	355
TsgcWSPServer_WAMP	356
TsgcWSPClient_WAMP	357
协议 WAMP JavaScript.....	358

订阅者.....	361
发布者.....	362
简单RPC	363
RPC进度结果.....	364
Protocol WAMP2	366
TsgcWSPClient_WAMP2	367
协议默认.....	369
TsgcWSPServer_sgc.....	370
TsgcWSPClient_sgc	371
TsgcIWWSPClient_sgc	372
协议默认 Javascript	373
数据集协议	377
TsgcWSPServer_Dataset	378
TsgcWSPClient_Dataset	379
TsgcIWWSPClient_Dataset.....	380
数据集协议 Javascript	381
协议数据集复制表.....	384
协议数据集通知更新	385
Protocol Files	386
TsgcWSPServer_Files.....	387
TsgcWSPClient_Files.....	388
TsgcWSMessageFile	389
如何向服务器发送文件	390
如何向客户端发送文件	391
如何发送大文件	392
Protocol Presence	393
TsgcWSPServer_Presence	394
TsgcWSPPresenceMessage	395
TsgcWSPClient_Presence.....	396
Protocol Presence Javascript	397
E2EE 协议.....	400

TsgcWSPServer_E2EE	402
TsgcWSPClient_E2EE	403
API	404
API Binance	406
Binance Connect WebSocket API	413
Binance 订阅 WebSocket 通道	414
Binance 获取市场数据	415
Binance 私有 REST API	416
Binance 现货交易	417
Binance 私有请求时间	419
Binance 提币	420
API Binance Futures	421
API Binance 期货交易	426
API SocketIO	427
API Coinbase	429
连接Coinbase WebSocket API	433
订阅Coinbase WebSocket频道	434
获取Coinbase市场数据	435
Coinbase私有REST API	436
Coinbase私有请求时间	437
Coinbase下单	438
Coinbase沙盒账户	439
API SignalRCore	440
API SignalR	446
API Kraken	449
API Kraken WebSockets Public	451
API Kraken WebSockets Private	457
API Kraken REST 公共接口	460
API Kraken REST 私有	462
API Kraken Futures	465
API Kraken Futures WebSockets 公开频道	467

API Kraken Futures WebSockets 私有	474
API Kraken Futures REST 公共接口	480
API Kraken Futures REST Private	482
API Pusher	487
API Bitmex	495
Bitmex 连接 WebSocket API	499
Bitmex 订阅 WebSocket 频道	500
如何下 Bitmex 订单	501
API Bitfinex	503
API Kucoin	507
Kucoin Connect WebSocket API	512
Kucoin 订阅 WebSocket 频道	513
Kucoin 获取市场数据	514
Kucoin 私有 REST API	515
Kucoin Trade Spot	516
Kucoin 私有请求时间	518
API Kucoin Futures	519
Kucoin Futures 连接 WebSocket API	524
Kucoin Futures 订阅 WebSocket 频道	525
Kucoin Futures Get Market Data	526
Kucoin Futures 私有 REST API	527
Kucoin 期货交易	528
Kucoin 期货私有请求时间	531
API 3Commas	532
API OKX	537
API XTb	542
API Bybit	545
API Forex	549
Forex Connect REST API	551
外汇交易订单	552
外汇账户查询	553

外汇行情数据	554
Forex Connect WebSocket API	555
Forex 订阅数据流	556
API Blockchain	558
API Cex.....	560
API Cex Plus	567
API Discord.....	571
API OpenAI	574
API MEXC	575
API MEXC Futures.....	579
API Bitget.....	582
API GateIO	584
API Deribit	586
API Crypto.com.....	589
API HTX	591
API Whatsapp	593
WhatsApp 创建应用程序	596
WhatsApp 手机号码 ID	597
WhatsApp Token	598
WhatsApp Webhook	599
WhatsApp 安全	600
WhatsApp 发送消息.....	601
WhatsApp 发送交互式消息.....	604
WhatsApp 发送模板消息	608
WhatsApp 接收消息和状态通知	610
WhatsApp 发送文件.....	612
WhatsApp 下载媒体.....	614
Telegram API.....	615
发送带内联按钮的 Telegram 消息.....	623
发送带按钮的 Telegram 消息	624
发送 Telegram 粗体文字消息	625

未找到作为机器人的 Telegram 聊天	626
Telegram 赞助消息	627
发送 Telegram 发票消息	628
Telegram 获取超级群组成员	629
添加 Telegram 代理	630
注册 Telegram 用户	631
RCON	632
CryptoHopper	633
RTCMultiConnection	637
WebPush	639
TsgcWSAPIServer_WebPush	640
TsgcWebPush_Client.....	641
扩展	642
PerMessage-Deflate	643
Deflate-Frame	644
MCP	645
TsgcWSAPIServer_MCP	646
MCP 服务器会话	648
MCP 服务器工具	649
MCP 服务器提示	652
MCP 服务器资源	655
MCP 服务器根目录	658
MCP 服务器采样	659
MCP Server Elicitation.....	660
TsgcWSAPIClient_MCP	661
MCP 客户端工具	662
MCP 客户端提示	664
MCP 客户端资源	666
MCP 客户端根目录	668
MCP 客户端采样	669
MCP 客户端引导 (Elicitation)	670

OpenAI.....	671
TsgcHTTP_API_OpenAI.....	678
OpenAI Completion	679
OpenAI 聊天	680
OpenAI Edit.....	681
OpenAI Audio.....	682
OpenAI 内容审核.....	683
OpenAI RealTime.....	684
OpenAI 响应	685
OpenAI 语音	686
OpenAI Fine-Tuning	687
OpenAI 批处理	688
OpenAI 上传	689
OpenAI 应用程序.....	690
TsgcAIOpenAIAssistant.....	691
TsgcAIOpenAIAssistant 文件搜索.....	693
TsgcAIOpenAIAssistant 流式传输.....	695
TsgcAIOpenAIAssistant 函数调用.....	697
OpenAI Audio.....	699
TsgcAudioRecorderMCI	700
TsgcAudioPlayerMCI	701
TsgcTextToSpeechSystem.....	702
TsgcTextToSpeechGoogle	703
TsgcTextToSpeechAmazon	704
TsgcAIChat.....	705
TsgcAIOpenAIChatBot	706
TsgcAIOpenAITranslator.....	707
TsgcAIOpenAIEmbeddings.....	708
TsgcAIDatabaseVectorFile.....	709
TsgcAIDatabaseVectorPinecone.....	710
嵌入创建向量	711

嵌入向量聊天机器人	712
Pinecone	713
Anthropic	716
TsgcHTTP_API_Anthropic	720
Anthropic Messages	721
Anthropic Vision	723
Anthropic 工具使用	725
Anthropic 模型	727
Anthropic 批处理	728
Anthropic 扩展思考	729
Anthropic 文档	731
Anthropic 提示缓存	733
Anthropic 引用	735
Anthropic Web Search	736
Anthropic 结构化输出	738
Anthropic Files	740
Anthropic MCP 连接器	742
Gemini	744
TsgcHTTP_API_Gemini	746
DeepSeek	747
TsgcHTTP_API_DeepSeek	748
DeepSeek 消息	749
DeepSeek Vision	750
DeepSeek 模型	751
Gemini	752
Gemini 消息	754
Gemini Vision	756
Gemini 模型	757
Gemini 结构化输出	758
Gemini 令牌计数	759
Gemini 嵌入	760

Gemini 工具使用	761
Ollama	763
TsgcHTTP_API_Ollama	765
Ollama 消息.....	766
Ollama 模型.....	767
Ollama 嵌入.....	768
Grok	769
TsgcHTTP_API_Grok	770
Grok 消息.....	771
Grok 视觉.....	772
Grok 模型.....	773
Mistral AI	774
TsgcHTTP_API_Mistral.....	776
Mistral 消息	777
Mistral Vision	778
Mistral 模型	779
Mistral 嵌入	780
IoT	781
IoT Amazon MQTT 客户端	782
IoT Azure MQTT 客户端.....	789
HTTP.....	794
HTTP2	795
TsgcHTTP2Client.....	796
请求 HTTP/2 方法	798
HTTP/2 服务器推送	799
HTTP/2 下载文件	800
HTTP/2 部分响应	801
HTTP/2 标头.....	802
客户端关闭连接	803
客户端保持连接活跃	804
HTTP/2 断线原因	805

客户端待处理请求.....	806
客户端认证.....	807
HTTP/2 与 OAuth2.....	808
TsgcHTTP2ConnectionClient.....	809
TsgcHTTP2RequestProperty.....	810
TsgcHTTP2ResponseProperty.....	811
Apple 推送通知.....	812
生成远程通知 APNs.....	813
向 APNs 发送通知请求.....	814
基于令牌的 APNs 连接.....	815
基于证书的 APNs 连接.....	816
HTTP1.....	818
OAuth2.....	822
TsgcHTTP_OAuth2_Client.....	823
适用于 Web 应用程序的 OAuth2 客户端.....	827
桌面应用程序的 OAuth2 客户端.....	828
TsgcHTTP_OAuth2_Client_Google.....	829
TsgcHTTP_OAuth2_Client_Microsoft.....	830
授权码授权（RFC 6749）.....	831
带 PKCE 的授权码流程（RFC 7636）.....	833
客户端凭据授权（RFC 6749）.....	835
Resource Owner Password Credentials Grant（RFC 6749）.....	836
设备授权许可（RFC 8628）.....	838
DPoP - 演示持有证明（RFC 9449）.....	840
TsgcHTTP_OAuth2_Server.....	844
OAuth2 服务器示例.....	847
OAuth2 自定义登录 HTML.....	851
OAuth2 服务器端点.....	852
OAuth2 注册应用程序.....	853
OAuth2 恢复访问令牌.....	854
OAuth2 服务器身份验证.....	855

OAuth2 无身份验证 URL.....	856
授权码许可	857
客户端凭据授权	859
资源所有者密码凭证授权	861
刷新令牌授权	863
设备授权许可 (RFC 8628)	865
令牌撤销 (RFC 7009)	867
令牌自省 (RFC 7662)	869
DPoP 验证 (RFC 9449)	871
TsgcHTTP_OAuth2_Server_Provider	873
OAuth2 提供商 Azure AD	875
OAuth2 提供商私有端点	876
OAuth2 提供程序身份验证	877
OAuth2 提供商请求.....	879
JWT	880
TsgcHTTP_JWT_Client	882
TsgcHTTP_JWT_Server.....	885
WebAuthn	886
TsgcWSAPIServer_WebAuthn	887
WebAuthn 注册.....	890
WebAuthn 注册请求	893
WebAuthn 注册响应	894
WebAuthn 注册结果	896
WebAuthn 身份验证	897
WebAuthn 身份验证请求.....	899
WebAuthn 身份验证响应.....	900
WebAuthn 认证结果	901
WebAuthn MDS	902
WebAuthn 授权.....	903
WebAuthn 授权 HTTP.....	904
WebAuthn 授权 WebSocket	905

Webauthn Javascript 客户端.....	906
Amazon SQS	910
Google OAuth2 密钥.....	915
Google 服务账号	921
Google Cloud Pub/Sub	925
Google Calendar.....	933
Google Calendar 同步日历.....	938
Google Calendar 同步事件.....	939
Google 日历刷新令牌.....	940
Google Calendar 服务账号.....	941
Google Cloud FCM.....	942
TsgcWebView2.....	945
WebView2 导航	947
WebView2 JavaScript.....	949
WebView2 Cookie.....	951
WebView2 下载	952
WebView2 设置	954
WebView2 高级功能.....	956
TsgcUDPCliant	959
TsgcUDPServer	960
STUN	961
TsgcSTUNClient	962
STUN 客户端 UDP 重传.....	964
STUN 客户端长期凭据.....	965
STUN 客户端属性	966
TsgcSTUNServer	967
STUN 服务器长期凭据.....	968
STUN 服务器备用服务器	969
TURN.....	970
TsgcTURNClient	971
TURN 客户端分配 IP 地址	973

TURN 客户端创建权限.....	974
TURN 客户端发送 Indication.....	975
TURN 客户端通道	976
TsgcTURNServer	977
TURN 服务器长期凭据.....	978
TURN Server Allocations.....	979
ICE	980
TsgcICEClient.....	981
ICE 收集候选者.....	982
ICE 配对候选	983
TsgcRTCPeerConnection	984
RTCPeerConnection WebSocket 服务器	986
RTCPeerConnection WebSocket 客户端	987
RTCPeerConnection STUN TURN	988
RTCPeerConnection 信令.....	989
RTCPeerConnection ICE.....	990
RTCPeerConnection DTLS	991
RTCPeerConnection 数据.....	992
Datasnap	993
TsgcWSHTTPWebBrokerBridgeServer	994
TsgcWSHTTP2WebBrokerBridgeServer	996
TsgcWSServer_HTTPAPI_WebBrokerBridge.....	997

OpenAPI 998

OpenAPI	998
OpenAPI 解析器 Pascal	1000
OpenAPI 附加属性	1005
OpenAPI 命令行	1007
OpenAPI 库.....	1009
OpenAPI API.....	1011
OpenAPI 客户端	1012

OpenAPI Server	1015
OpenAPI 服务器规范优先	1017
OpenAPI 服务器代码优先	1018
OpenAPI Amazon AWS	1019
OpenAPI Amazon AWS 凭证	1025
OpenAPI Amazon AWS S3	1027
OpenAPI Google Cloud	1028
OpenAPI Google Cloud OAuth2	1033
OpenAPI Google Cloud 服务账号	1036
OpenAPI Google Cloud PubSub	1040
OpenAPI Google Cloud Calendar	1041
OpenAPI Google Drive	1042
OpenAPI Microsoft	1044
OpenAPI Microsoft Tenant	1049
OpenAPI Microsoft 注册应用程序	1050
OpenAPI Microsoft OAuth2 代码	1053
OpenAPI Microsoft OAuth2 凭据	1055
OpenAPI Microsoft Graph	1057
API	1058
AbstractApi 地理位置	1059
演示	1060
服务器聊天	1060
客户端聊天	1062
Client	1063
Client MQTT	1064
Client SocketIO	1066
服务器监控	1067
服务器快照	1069
客户端快照	1070
上传文件	1071

服务器身份验证	1073
KendoUI_Grid.....	1074
ServerSentEvents	1076
服务器 WebRTC.....	1077
AppRTC 服务器.....	1078
Telegram 客户端	1080
第三方	1082
Coturn.....	1082
许可证	1084
许可证.....	1084
索引	1086
PDF-Back-Cover	1094

简介

sgcWebSockets 是一个专业级组件库，用于实时通信、消息传递和 AI 集成。它提供了 WebSockets、HTTP/2、MQTT、AMQP、WebRTC、Server-Sent Events 以及 30 余种第三方 API 连接器的生产就绪实现，所有这些均通过一致的事件驱动组件架构访问。无论您是在构建交易平台、IoT 网关、AI 驱动应用程序还是企业消息系统，sgcWebSockets 都能为您提供所需的网络基础。

该库支持 **Delphi 7 至 Delphi 13**、**C++Builder**、**Lazarus/FreePascal** 以及 **.NET** (Framework 2.0+、.NET Core 1.0+、.NET 5–9、.NET Standard)。应用程序可以使用 VCL 和 FireMonkey 从单一代码库面向 Windows、macOS、Linux、iOS 和 Android。

Delphi / C++Builder / FreePascal 特性

核心 WebSocket

- 功能完整的**多线程 WebSocket 服务器**，实现了 **RFC 6455**。
- **Text** 和 **Binary** 消息支持。
- 消息压缩，使用 **PerMessage_Deflate** 扩展 (**RFC 7692**)。
- 内置 **WatchDog** 和 **HeartBeat**，用于自动重连和连接监控。
- **负载均衡服务器**，用于将连接分发到多个后端。
- **代理服务器**组件，允许 Web 浏览器连接至任意 TCP 服务器。
- 通过 **HTTP 代理**和 **SOCKS 代理服务器**的客户端连接。
- 事件：OnConnect、OnDisconnect、OnMessage、OnError、OnHandshake。

高性能服务器

- **Microsoft HTTP 服务器 API** (HTTP.SYS) 与 **IOCP** 结合，用于高性能 Windows 服务器。
- **Indy 服务器**支持 **IOCP** (Windows)、**EPOLL** (Linux) 或默认的每连接一线程模型。
- **WebSocket** 和 **HTTP/2** 连接使用同一端口。
- **WebBroker 服务器**，在单一端口上支持 **DataSnap**、**HTTP/2** 和 **WebSocket**。
- 基于 **WinHTTP API** 的客户端 **WebSocket**。

安全性和身份验证

- **SSL/TLS** 适用于服务器和客户端组件。支持 **OpenSSL 1.1.1** 和 **3.0**。客户端在 Windows 上支持 **SChannel**。
- **OAuth 2.0** 和 **JWT** (JSON Web Token) 客户端和服务组件。
- **WebAuthn 服务器协议**，用于基于通行密钥的认证。
- **E2EE** (端到端加密) 子协议。

AI 与机器学习

- **OpenAI** 和 **Claude** 与 **MCP Server** 和 **MCP Client** 组件的集成。
- **ChatBot** 和 **翻译器**应用程序组件，支持语音命令。
- **嵌入和向量数据库**支持 (基于文件和 Pinecone)。
- **文字转语音**，通过系统默认、Google Cloud 和 Amazon AWS 实现。

HTTP/2 协议

- 完整的 **HTTP/2 客户端和服务器**实现。
- **Server-Sent Events** (推送通知) 通过 HTTP 传输。

消息传递协议

- **MQTT 3.1.1** 和 **5.0**，支持 **WebSocket** 和普通 TCP。
- **AMQP 0.9.1** 和 **1.0**，支持 **RabbitMQ broker**。
- **STOMP** 通过 **WebSocket** 和 TCP (**ActiveMQ**、**RabbitMQ 代理**)。
- **WAMP 1.0** 和 **2.0** 客户端和服务协议。
- **JSON-RPC 2.0** 子协议。

P2P 和实时通信

- **WebRTC** 服务器协议。
- **STUN** 和 **TURN** 客户端和服务组件。
- **ICE NAT** 穿越客户端。
- **UDP** 客户端和服务。

API 集成

- **30 余个内置 API 连接器**：Binance、Coinbase、Kraken、Bitfinex、Discord、WhatsApp、Telegram 等。
- **SignalR** 和 **SignalR Core**、**Socket.IO** 以及 **Pusher** 客户端。
- 云服务：**Amazon SQS**、**Google Cloud Pub/Sub**、**Google Calendar**。

跨平台

- **VCL** 和 **FireMonkey** (Windows、macOS、iOS、Android、Linux)。
- **C++Builder** 支持。
- **Lazarus / FreePascal** 支持。
- 浏览器客户端的内置 **JavaScript 库**。

以下组件包含在 sgcWebSockets 库中：

1 sgcWebSockets

- **TsgcWebSocketClient** : 基于 Indy 库的 WebSocket 客户端。
- **TsgcWebSocketServer** : 基于 Indy 库的 WebSocket 服务器。
- **TsgcWebSocketHTTPServer** : 基于 Indy 库的 WebSocket + HTTP 服务器。
- **TsgcWebSocketServer_HTTPAPI** : 基于 HTTP.SYS Microsoft HTTP API 的高性能 WebSocket + HTTP 服务器。
- **TsgcWebSocketClient_WinHTTP** : 基于 WinHTTP 库的 WebSocket 客户端。
- **TsgcWebSocketFirewall** : 具有 IP 过滤、暴力破解保护、速率限制和威胁检测功能的 WebSocket 服务器防火墙。
- **TsgcWebSocketLoadBalancerServer** : 用于 WebSocket 和 HTTP 协议的负载均衡服务器。
- **TsgcWebSocketProxyServer** : 允许 Web 浏览器连接到 TCP 服务器的 WebSocket 代理服务器。

2 sgcWebSocket API

- **TsgcWSAPI_Binance** : Binance 现货客户端, 支持 WebSocket + REST API。
- **TsgcWSAPI_Binance_Futures** : Binance 期货客户端, 支持 WebSocket + REST API。
- **TsgcWSAPI_SocketIO** : Socket.IO 客户端。
- **TsgcWSAPI_Coinbase** : Coinbase Pro 客户端, 支持 WebSocket + REST API。
- **TsgcWSAPI_Bitmex** : Bitmex 客户端, 支持 WebSocket + REST API。
- **TsgcWSAPI_SignalR** : SignalR WebSocket 客户端。
- **TsgcWSAPI_SignalRCore** : SignalRCore WebSocket 客户端。
- **TsgcWSAPI_Pusher** : Pusher WebSocket 客户端。
- **TsgcWSAPI_Kraken** : Kraken 客户端 API, 支持 WebSocket 和 REST API。
- **TsgcWSAPI_Kraken_Futures** : Kraken 期货客户端 API, 支持 WebSocket 和 REST API。
- **TsgcWSAPI_Bitstamp** : Bitstamp WebSocket 客户端。
- **TsgcWSAPI_Cex** : Cex WebSocket 客户端。
- **TsgcWSAPI_Huobi** : Huobi WebSocket 客户端。
- **TsgcWSAPI_ThreeCommas** : ThreeCommas 客户端 API。
- **TsgcWSAPI_Bitfinex** : Bitfinex WebSocket API。
- **TsgcWSAPI_Discord** : Discord WebSocket 客户端。
- **TsgcWSAPI_BlockChain** : BlockChain WebSocket 客户端。
- **TsgcWSAPI_Bybit** : Bybit WebSocket 客户端。
- **TsgcWSAPI_OKX** : OKX WebSocket 客户端。
- **TsgcWSAPI_Kucoin** : KuCoin 现货 WebSocket 客户端。
- **TsgcWSAPI_Kucoin_Futures** : KuCoin 期货 WebSocket 客户端。
- **TsgcWSAPI_Deribit** : Deribit WebSocket 客户端。
- **TsgcWSAPI_CryptoCom** : Crypto.com WebSocket 客户端。
- **TsgcWSAPI_HTX** : HTX WebSocket 客户端。
- **TsgcWSAPI_MEXC** : MEXC WebSocket 客户端。
- **TsgcWSAPI_Bitget** : Bitget WebSocket 客户端。
- **TsgcWSAPI_GateIO** : Gate.io WebSocket 客户端。
- **TsgcWSAPI_XTB** : XTB WebSocket 客户端。
- **TsgcWSAPI_CexPlus** : CexPlus WebSocket 客户端。

- [TsgcWSAPI_OpenAI](#) : OpenAI 实时 WebSocket API 客户端。

3 sgcWebSocket 库

- [TsgcTDLlib_Telegram](#) : Telegram API 客户端。
- [TsgcWhatsApp_Client](#) : WhatsApp 商业云客户端。
- [TsgcHTTP_Cryptohopper](#) : Cryptohopper 客户端 API。
- [TsgcLib_RCON](#) : RCON 客户端。

4 sgcWebSocket 协议

- [TsgcWSPClient_MQTT](#) : MQTT (3.1.1 和 5.0) 客户端。支持 WebSocket 和普通 TCP 连接。
- [TsgcWSPClient_AMQP1](#) : AMQP 1.0.0 客户端。支持 RabbitMQ Broker。
- [TsgcWSPClient_AMQP](#) : AMQP 0.9.1 客户端。支持 RabbitMQ Broker。
- [TsgcWSPClient_STOMP](#) : STOMP 客户端, 支持 WebSocket 和普通 TCP 连接。
 - [TsgcWSPClient_STOMP_ActiveMQ](#) : ActiveMQ Broker 的 STOMP 客户端。
 - [TsgcWSPClient_STOMP_RabbitMQ](#) : RabbitMQ Broker 的 STOMP 客户端。
- [TsgcWSPClient_WAMP](#) : WAMP 1.0 客户端协议。
- [TsgcWSPServer_WAMP](#) : WAMP 1.0 服务器协议。
- [TsgcWSPClient_WAMP2](#) : WAMP 2.0 客户端协议。
- [TsgcWSPServer_AppRTC](#) : 基于 AppRTC Google 项目的 WebRTC 服务器。
- [TsgcWSPServer_WebRTC](#) : WebRTC 服务器协议。
- [TsgcWSPClient_sgc](#) : 基于 JSON RPC 的 WebSocket 客户端 SGC 协议。
- [TsgcWSPServer_sgc](#) : 基于 JSON RPC 的 WebSocket 服务器 SGC 协议。
- [TsgcWSPClient_Files](#) : WebSocket 文件传输客户端协议。
- [TsgcWSPServer_Files](#) : WebSocket 文件传输服务器协议。
- [TsgcWSPClient_Dataset](#) : WebSocket 客户端数据集同步协议。
- [TsgcWSPServer_Dataset](#) : WebSocket 服务器数据集同步协议。
- [TsgcWSPClient_Presence](#) : WebSocket 客户端在线状态协议。
- [TsgcWSPServer_Presence](#) : WebSocket 服务器在线状态协议。
- [TsgcWSPClient_E2EE](#) : 端到端加密客户端协议。
- [TsgcWSPServer_E2EE](#) : 端到端加密服务器协议。

5 sgcWebSockets HTTP

- [TsgcHTTP1Client](#) : 基于 Indy TIdHTTP 的 HTTP 1.0 客户端。
- [TsgcHTTP2Client](#) : HTTP 2.0 客户端。
- [TsgcHTTP_JWT_Client](#) : JWT (JSON Web Token) 客户端。
- [TsgcHTTP_JWT_Server](#) : JWT (JSON Web Token) 服务器。
- [TsgcHTTP_OAuth2_Client](#) : OAuth 2.0 客户端。
- [TsgcHTTP_OAuth2_Server](#) : OAuth 2.0 服务器。
- [TsgcHTTPAWS_SQS_Client](#) : Amazon AWS SQS 客户端。
- [TsgcHTTPGoogleCloud_PubSub_Client](#) : Google Cloud Pub/Sub 客户端。
- [TsgcHTTPGoogleCloud_Calendar_Client](#) : Google Calendar 客户端。
- [TsgcHTTP_OAuth2_Client_Google](#) : 适用于 Google 的 OAuth 2.0 客户端。
- [TsgcHTTP_OAuth2_Client_Microsoft](#) : 适用于 Microsoft 的 OAuth 2.0 客户端。

- [TsgcHTTP_OAuth2_Server_Provider](#) : OAuth 2.0 服务器提供程序。
- [TsgcWSAPIServer_WebAuthn](#) : 基于通行密钥身份验证的 WebAuthn 服务器。
- [TsgcHTTPGoogleCloud_FCM_Client](#) : Google Cloud FCM (Firebase Cloud Messaging) 客户端。

6 sgcWebSockets IoT

- [TsgcIoTAmazon_MQTT_Client](#) : Amazon MQTT IoT Core 客户端。
- [TsgcIoTAzure_MQTT_Client](#) : Azure IoT MQTT 客户端。

7 sgcWebSockets P2P

- [TsgcUDPClient](#) : UDP 客户端。
- [TsgcUDPServer](#) : UDP 服务器。
- [TsgcSTUNClient](#) : STUN 客户端。
- [TsgcSTUNServer](#) : STUN 服务器。
- [TsgcTURNClient](#) : STUN / TURN 客户端。
- [TsgcTURNServer](#) : STUN / TURN 服务器。
- [TsgcICEClient](#) : ICE 客户端。
- [TsgcRTCPeerConnection](#) : WebRTC 对等连接客户端。

8 sgcWebSockets DataSnap

- [TsgcWSHTTPWebBrokerBridgeServer](#) : 支持 HTTP + WebSocket 的 DataSnap 服务器替代组件。
- [TsgcWSHTTP2WebBrokerBridgeServer](#) : 支持 HTTP + HTTP/2 + WebSocket 的 DataSnap 服务器替代组件。
- [TsgcWSServer_HTTPAPI_WebBrokerBridge](#) : 基于 HTTP.SYS Microsoft 服务器的 DataSnap 服务器替代组件。

9 sgcWebSockets AI

- [TsgcAIOpenAIChatBot](#) : 构建带语音命令的聊天机器人。
- [TsgcAIOpenAITranslator](#) : 实时翻译。
- [TsgcAudioRecorderMCI](#) : 使用 MCI 录制音频。
- [TsgcAudioPlayerMCI](#) : 使用 MCI 播放音频。
- [TsgcTextToSpeechSystem](#) : 使用操作系统默认设置进行文本转语音。
- [TsgcTextToSpeechGoogle](#) : 使用 Google Cloud 进行文本转语音。
- [TsgcTextToSpeechAmazon](#) : 使用 Amazon AWS 进行文本转语音。
- [TsgcAIOpenAIEmbeddings](#) : 允许您使用自定义数据构建 AI 应用程序。
- [TsgcAIDatabaseVectorFile](#) : 将向量存储在纯文本文件中。
- [TsgcAIDatabaseVectorPinecone](#) : 支持 Pinecone 向量数据库。
- [TsgcAIOpenAIAssistant](#) : OpenAI Assistants API 客户端。
- [TsgcWSAPIServer_MCP](#) : MCP (模型上下文协议) 服务器。
- [TsgcWSAPIClient_MCP](#) : MCP (模型上下文协议) 客户端。

版本支持

支持的 Delphi IDE

- Delphi 7 (*仅在升级到 Indy 10 后支持，不支持 Intraweb)
- 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 支持的 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

支持 FreePascal 的 IDE

- Lazarus

Trial 版本

免费版提供的已编译 *.dcu 文件使用默认的 Indy 和 Intraweb 版本。如果您已升级这些软件包，它们可能无法正常工作，您需要购买完整源代码版本。

Indy 软件包

部分组件使用 Indy 作为其 TCP/IP 库（如 TsgcWebsocketClient 或 TsgcWebsocketServer），这意味着安装 sgcWebSockets 包需要 Indy。默认情况下，sgcWebSockets 使用内置于 RAD Studio 的 Indy 库，但我们提供了一个具有更多功能的自定义 Indy 版本：支持 OpenSSL API 1.1、OpenSSL 3.0、ALPN 协议等。

安装

Delphi / CBuilder / Lazarus

1. 将包含的文件解压到目录 { \$DIR }

2. 从 Delphi\CBuilder :

将解压文件所在的目录 { \$DIR } 添加到 Delphi/CBuilder 的库路径，路径位于"工具"→"环境选项"→"目录"下。

所有 Delphi\CBuilder 版本

将目录 { \$DIR }\source 添加到库路径

适用于特定 Delphi 版本

Delphi 7 : 将目录 { \$DIR }\libD7 添加到库路径中
 Delphi 2007 : 将目录 { \$DIR }\libD2007 添加到库路径
 Delphi 2009 : 将目录 { \$DIR }\libD2009 添加到库路径
 Delphi 2010 : 将目录 { \$DIR }\libD2010 添加到库路径
 Delphi XE : 将目录 { \$DIR }\libDXE 添加到库路径
 Delphi XE2 : 将目录 { \$DIR }\libDXE2\\$(Platform) 添加到库路径
 Delphi XE3 : 将目录 { \$DIR }\libDXE3\\$(Platform) 添加到库路径
 Delphi XE4 : 将目录 { \$DIR }\libDXE4\\$(Platform) 添加到库路径
 Delphi XE5 : 将目录 { \$DIR }\libDXE5\\$(Platform) 添加到库路径
 Delphi XE6 : 将目录 { \$DIR }\libDXE6\\$(Platform) 添加到库路径
 Delphi XE7 : 将目录 { \$DIR }\libDXE7\\$(Platform) 添加到库路径
 Delphi XE8 : 将目录 { \$DIR }\libDXE8\\$(Platform) 添加到库路径
 Delphi 10 : 将目录 { \$DIR }\libD10\\$(Platform) 添加到库路径
 Delphi 10.1 : 将目录 { \$DIR }\libD10_1\\$(Platform) 添加到库路径中
 Delphi 10.2 : 将目录 { \$DIR }\libD10_2\\$(Platform) 添加到库路径
 Delphi 10.3 : 将目录 { \$DIR }\libD10_3\\$(Platform) 添加到库路径
 Delphi 10.4 : 将目录 { \$DIR }\libD10_4\\$(Platform) 添加到库路径
 Delphi 11 : 将目录 { \$DIR }\libD11\\$(Platform) 添加到库路径中
 Delphi 12 : 将目录 { \$DIR }\libD12\\$(Platform) 添加到库路径
 Delphi 13 : 将目录 { \$DIR }\libD13\\$(Platform) 添加到库路径

针对特定 CBuilder 版本

C++ Builder 2010 : 将目录 { \$DIR }\libD2010 添加到库路径
 C++ Builder XE : 将目录 { \$DIR }\libDXE 添加到库路径
 C++ Builder XE2 : 将目录 { \$DIR }\libDXE2\\$(Platform) 添加至库路径
 C++ Builder XE3 : 将目录 { \$DIR }\libDXE3\\$(Platform) 添加到库路径
 C++ Builder XE4 : 将目录 { \$DIR }\libDXE4\\$(Platform) 添加到库路径
 C++ Builder XE5 : 将目录 { \$DIR }\libDXE5\\$(Platform) 添加到库路径
 C++ Builder XE6 : 将目录 { \$DIR }\libDXE6\\$(Platform) 添加到库路径
 C++ Builder XE7 : 将目录 { \$DIR }\libDXE7\\$(Platform) 添加到库路径
 C++ Builder XE8 : 将目录 { \$DIR }\libDXE8\\$(Platform) 添加到库路径
 C++ Builder 10 : 将目录 { \$DIR }\libD10\\$(Platform) 添加到库路径
 C++ Builder 10.1 : 将目录 { \$DIR }\libD10_1\\$(Platform) 添加到库路径
 C++ Builder 10.2 : 将目录 { \$DIR }\libD10_2\\$(Platform) 添加到库路径
 C++ Builder 10.3 : 将目录 { \$DIR }\libD10_3\\$(Platform) 添加到库路径
 C++ Builder 10.4 : 将目录 { \$DIR }\libD10_4\\$(Platform) 添加到库路径
 C++ Builder 11 : 将目录 { \$DIR }\libD11\\$(Platform) 添加到库路径
 C++ Builder 12 : 将目录 { \$DIR }\libD12\\$(Platform) 添加到库路径
 C++ Builder 13 : 将目录 { \$DIR }\libD13\\$(Platform) 添加到库路径

对于所有 CBuilder 版本，将 dcp\\$(Platform) 添加到库路径（包含 .bpi 文件）

3. 来自 Delphi

选择

通过 File（文件）菜单打开并浏览正确的 Packages\sgcWebSockets.groupproj（请先编译 sgcWebSocketsX.dpk，然后安装 dclsgcWebSocketsX.dpk）。

Delphi 的包文件

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. 来自 CBuilder

选择

通过 File（文件）菜单打开并浏览正确的 Packages\sgcWebSockets.groupproj（请先编译 sgcWebSocketsX.dpk，然后安装 dclsgcWebSocketsX.dpk）。

CBuilder 的软件包文件

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. 来自 Lazarus

选择：文件 > 打开，浏览至 Packages\sgcWebSocketsLazarus.lpk（先编译，再安装）

编译后的文件位于 Lazarus 目录中。该目录内有一个 Indy 目录，其中包含最新的 Indy 源代码版本。

已使用 Lazarus 2.0.6 和 Indy 10.5.9.4930 进行测试

6. 演示

所有演示均位于 Demos 子目录中。只需打开项目并运行即可。由于不同 Intraweb 版本的差异，IntraWeb 演示可能需要修改某些单元。

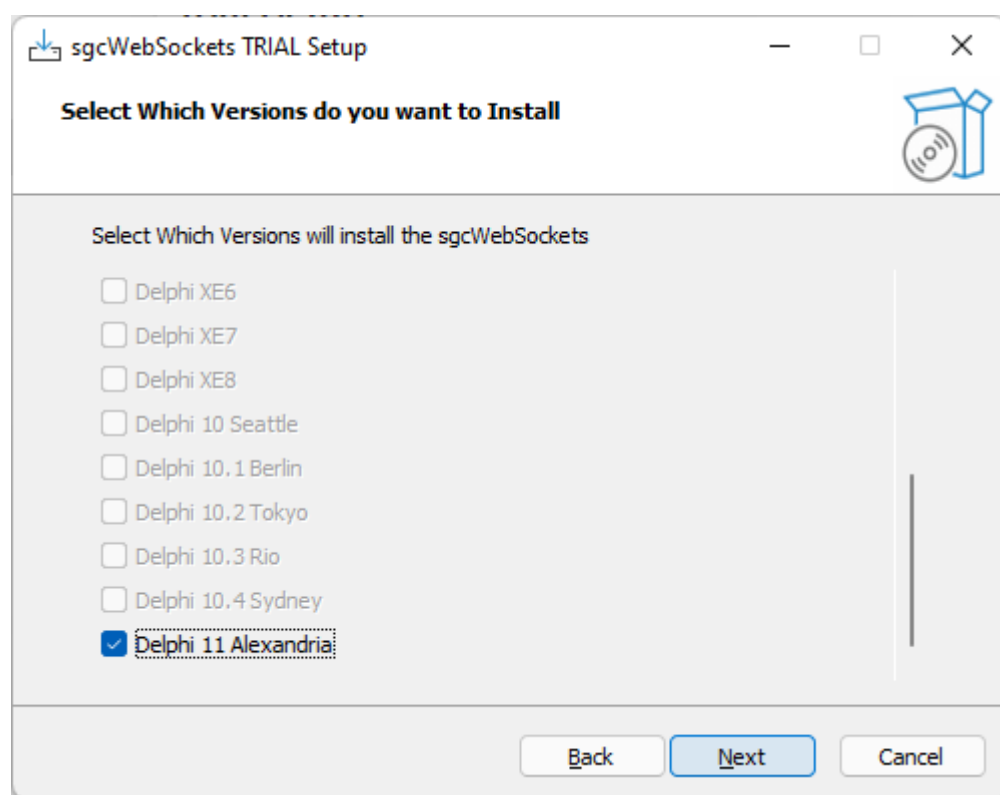
安装设置

*最低要求 Windows Vista（不支持 Windows 2000、XP 和 Server 2003）。

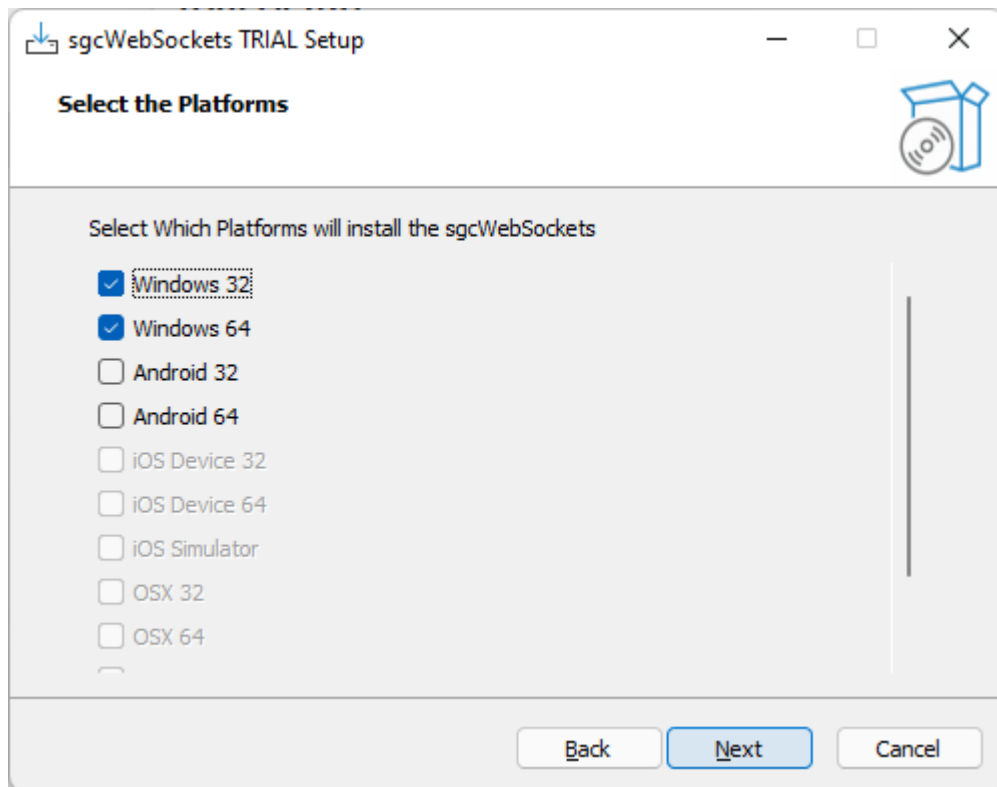
如果您使用 Windows 安装程序安装 sgcWebSockets 库，安装过程有向导引导，非常简单。若安装过程中出现任何错误，请参阅 [安装错误](#) 页面，也可尝试 [手动安装包](#)。

试用版设置

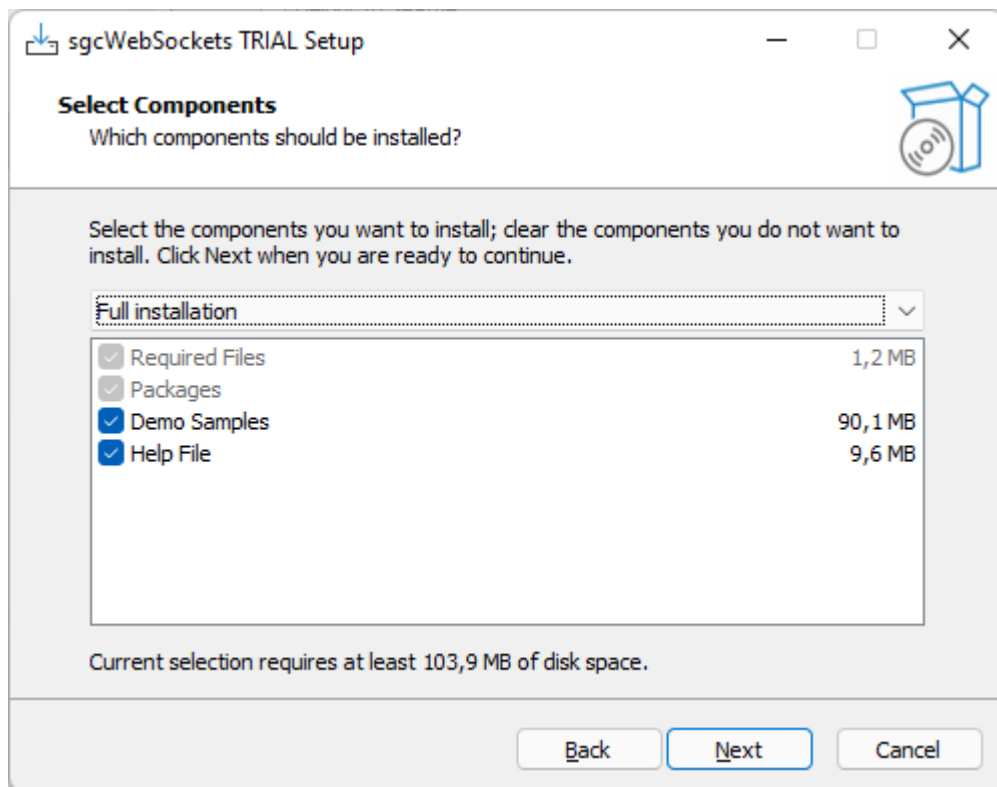
- 运行试用版安装程序。
- Trial 安装程序需要管理员权限。
- 安装程序将显示 Delphi / CBuilder / Rad Studio 版本列表，默认启用已下载的版本。如果安装程序未检测到该版本，将提取文件但不尝试编译。请参阅[手动安装包](#)。



- 下一页显示可安装的平台，仅启用安装程序检测到的平台。



- 下一页显示许可协议，安装 Trial 版本必须接受该协议。
- 接受许可协议后，将显示要安装的组件。默认情况下，包、已编译的 DCU、演示程序和帮助文件均将被安装。您可以自定义是否安装帮助文件和演示程序。

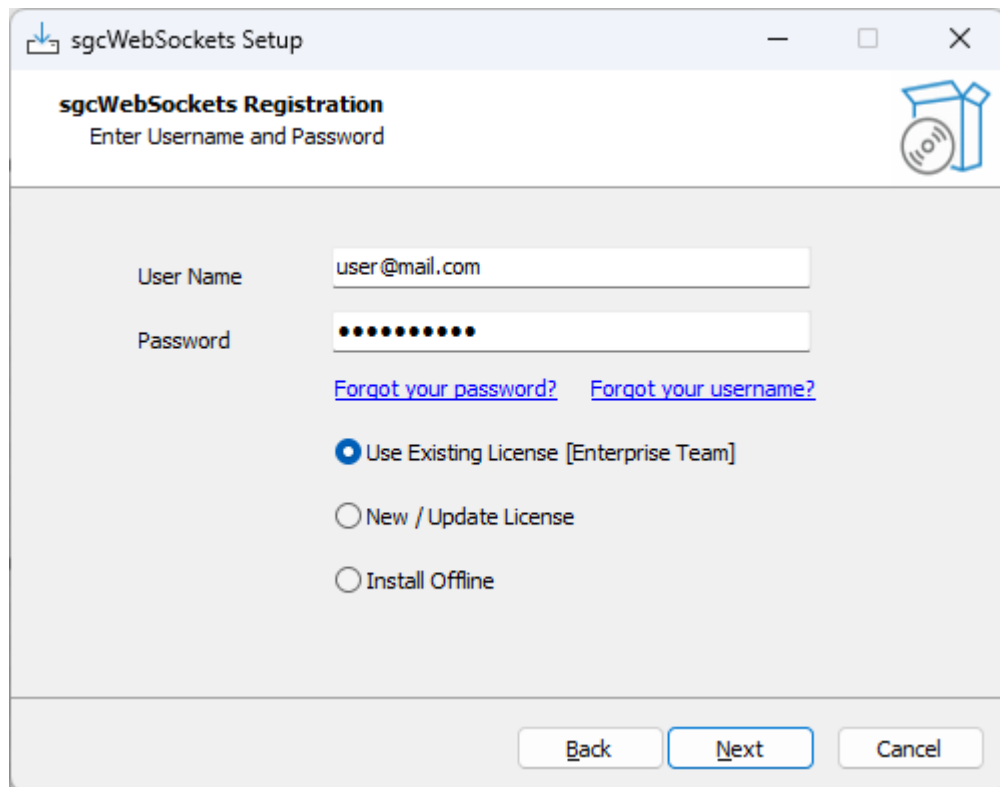


- 最后，它将提取文件、编译并安装包，并在 IDE 中注册所需路径。

客户设置

已购买许可证的用户可以使用安装程序安装 sgcWebSockets 库。以下是安装软件包的分步说明。

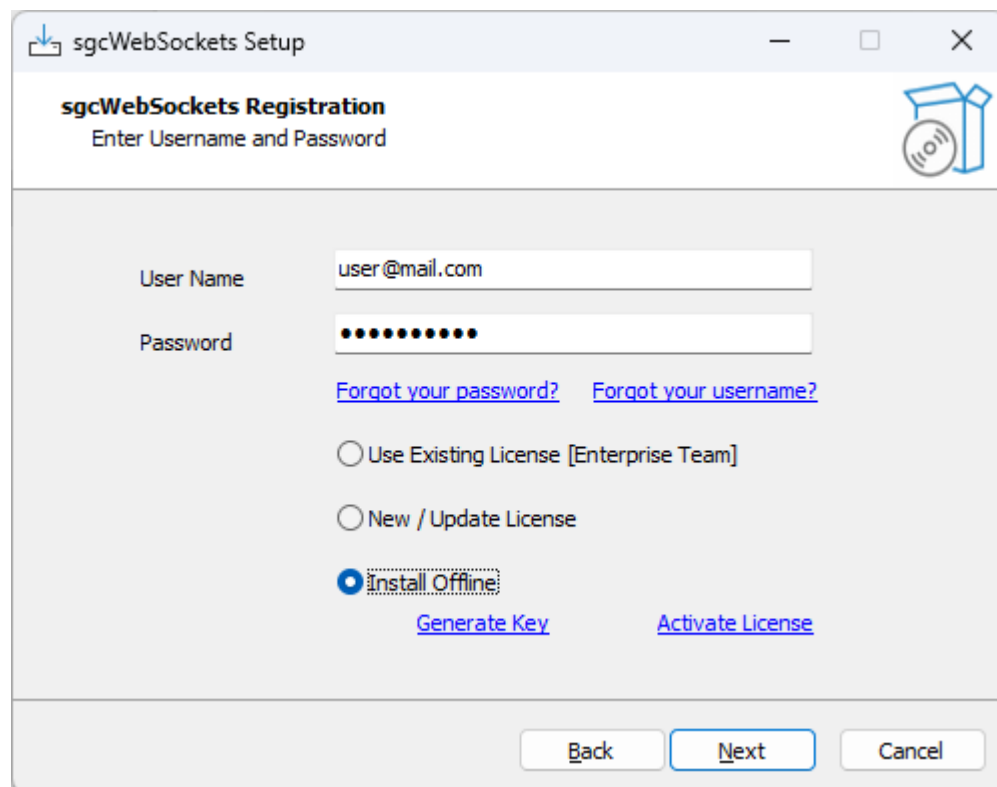
- 执行安装程序。
- 安装程序以最低权限运行（若以管理员身份运行，则无法安装到网络驱动器）。若目标位置需要管理员权限，请以管理员身份运行安装程序。
- 首先，您必须设置您私有 eSeGeCe 账户的用户名/密码。这只需输入一次；下次使用安装程序时，将读取之前保存的值。

The image shows a Windows-style window titled "sgcWebSockets Setup". Inside the window, the title "sgcWebSockets Registration" is displayed above the instruction "Enter Username and Password". On the right side of the window, there is a small icon of a box with a coin. The main area of the window contains two input fields: "User Name" with the text "user@mail.com" and "Password" with masked characters (dots). Below these fields are two links: "Forgot your password?" and "Forgot your username?". Underneath the links are three radio button options: "Use Existing License [Enterprise Team]" (which is selected), "New / Update License", and "Install Offline". At the bottom of the window, there are three buttons: "Back", "Next", and "Cancel".

- 共有 3 个选项：
 - **使用现有许可证**：如果此版本已安装，该选项将默认选中。它将使用此版本的最新配置。
 - **新建/更新许可证**：如果此版本之前未安装，该选项将默认选中。它将连接到许可证服务器以获取许可证信息。如果您最近升级了许可证，可选择此选项以更新许可证。
 - **离线安装**：如果计算机没有互联网访问，请选择此选项以激活您的许可证。

生成密钥

此选项生成一个将用于激活许可证的密钥。



sgcWebSockets Registration
Enter Username and Password

User Name:

Password:

[Forgot your password?](#) [Forgot your username?](#)

☐ Use Existing License [Enterprise Team]
☐ New / Update License
☒ **Install Offline**

[Generate Key](#) [Activate License](#)

Back Next Cancel

复制密钥并访问您的私人在线账户：www.esegece.com/my-account/subscriptions。



Generate Key - eSeGeCe Software (2023.8.0)

```
VjJ4b2MxTXlVWghpUm1oVltdHdZVlpxU2xOak1VNvIZMGhhYUZZd01UVlhhMIzVmpBeGMxZHVRbGRXUldzeFdWY3hSbVZXV25GUmJYQnNWbGQ
OTmxVeFZrOVRNREZ6WTBac1ZHSnVRbTlXym5CelRWWk9XR05JV21GTlIzaDRWVEZvYTFSDfJuSk5WRVpWVVFVd2QxcEVRbk5qUjFGNVdrVnd
hVlI6YUhsVk1XUjNaR3h2ZDJKSvJ3sUlHSM2hvVld0YWQyTnNaRmhOVmtwT1RvAG9XVlI4VWt0WlZscEdVbXBDVldFeFNsQlphMVV4VmtVeFdHskZO
VmRTVlhCM1ZrVmFVMU5yTVZaTlNHeFRWa1phVUZsWE1VOUSNVkoxWTBoT2FWWlVSVEZWTVdoUfUyeEZkMk6YUZwbGEzQlIxZjFprUzJSR1Nu
VIZiVpXVZzhMlF3VFRGU2NsVnJTazlTVkZaV1ZteG9hMVZHV2taU2FsSlZWbFpLZfZSc1ZYaFNWa1pWVld4Q1YxSXpVWHBxUkVaVFVXMVdjazFW
Vm1GbGEwcFBXVmQwUmsxR1VYaFZiSEJyVFVSc1ZsVldVbGRVYkVwSVpFuk9WVTFHU2tOVWExVTFVbFpTV1ZwR1FsTlNNRFF4VmtaV1UySnJN
WEpQVmxalU1ZrWktVRmxYTVU5Tk1WRjRwV3RhYTAxRVJrWlplWRTV2VkrKR2NsSlVSbFZXykVweFdrUkJOVlpXU2xWVGF6VlRvaksSTUZaSE1Y
ZFJheIZXVZVWVldGEZhbEZWYTFwV1RWWlNWbFZ0TlU1aGVsWldWVlpTVTFReVJuUmtSRVWVFVaS1lWUnNWWGhTVm5BMlUyeENVMUpWVl
hkV1J6rJNzbTFSZDA5VlZsaFdSWEJQVld0a2FtVldVWGRWYTbwUFVsUlZnBfZyYUd0VlJrcElWR3BPVkJaNVVqTlZSa1U1VUZFOVBRPT0=
```

选择要激活的订阅并粘贴密钥。

Offline License

```
TKVWSyprGK3VVH10ZtOVWEXVH1V0PpTVZWRK1SHNNK14Vmtav10yShnWVpQVmxao1Z1VWkVkmX
YTUVU5Tk1WRjRWV3RHYTAxRVJrWlpWRTV2VkrKR2NsSIVSbFZXyKVweFdrUKJOVlpXU2xWVGF6VIRVak5S
TUZaSE1YZFJhelZXVFZWV1dGEHvEbZWYTFwV1RWWINWbFZ0TIU1aGVsWldWVlpTVTFReVJuUmtSRTV
WVFVaS1WUnNWWGhTVm5BMLUyeENVMUUpWVlhKv1J6RjNZbTFSZDA5VIZsaFdSWEJQVld0a2FtVldVblJqUI
VwUFVsUldRmxVVG10VJrcEIWR3BhVkZaNIvqTIZSa1U1VUZFOVBRPT0=
```

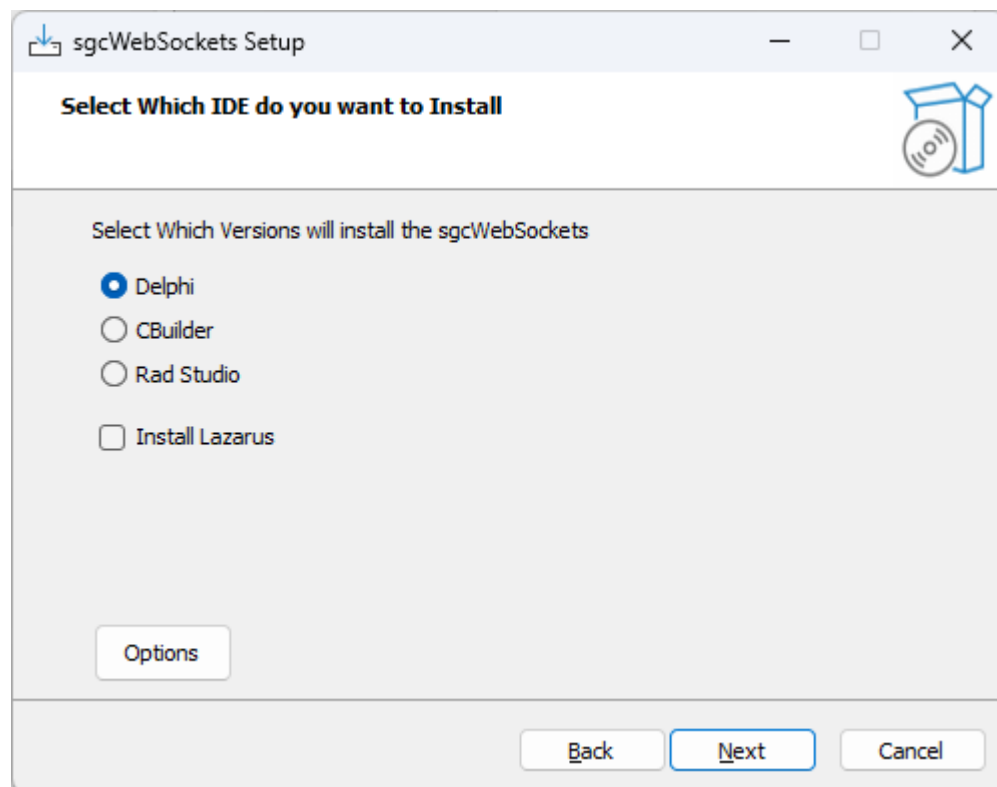
Activate License

激活许可证

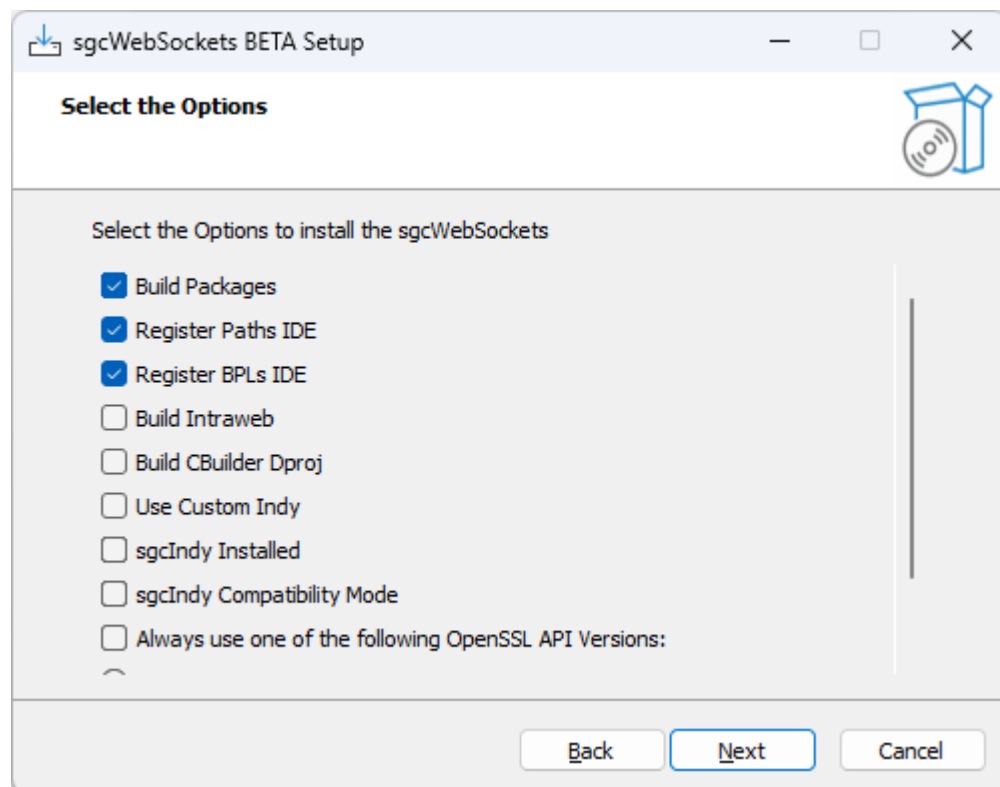
若请求正确，将返回一个许可证，该许可证必须复制到安装程序中。



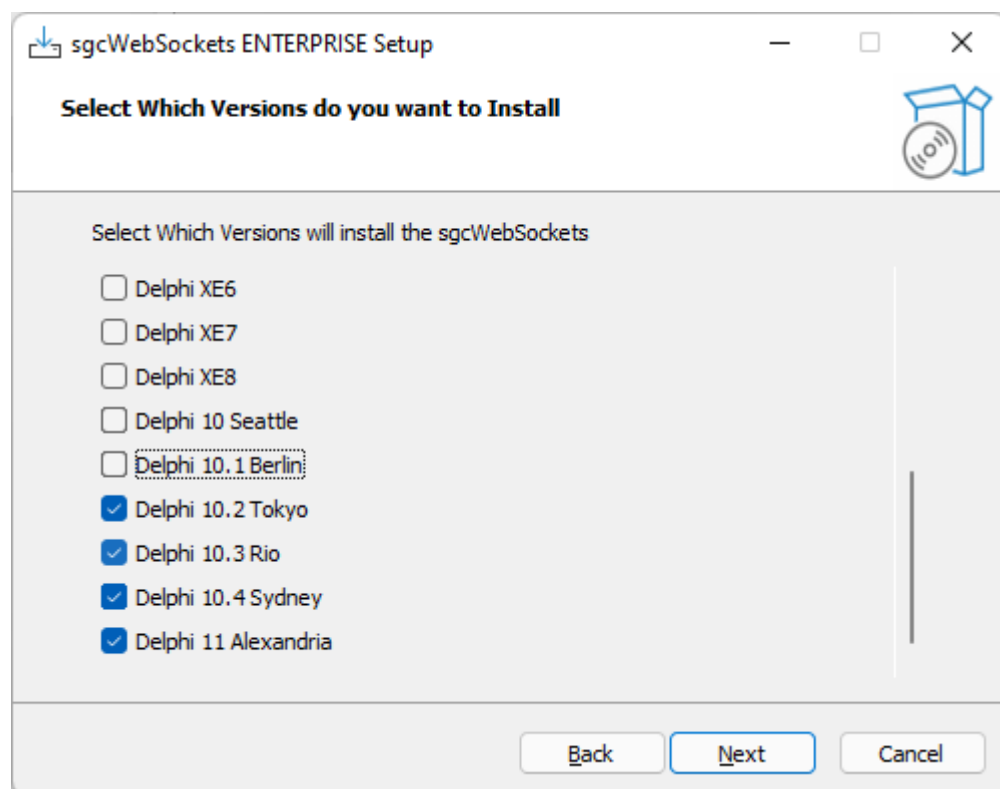
- 如果许可证激活成功，请选择是否要在 Delphi、CBuilder 或 Rad Studio IDE 中安装。有一个复选框用于提取所需的 Lazarus 文件（Lazarus 需要手动安装包）。



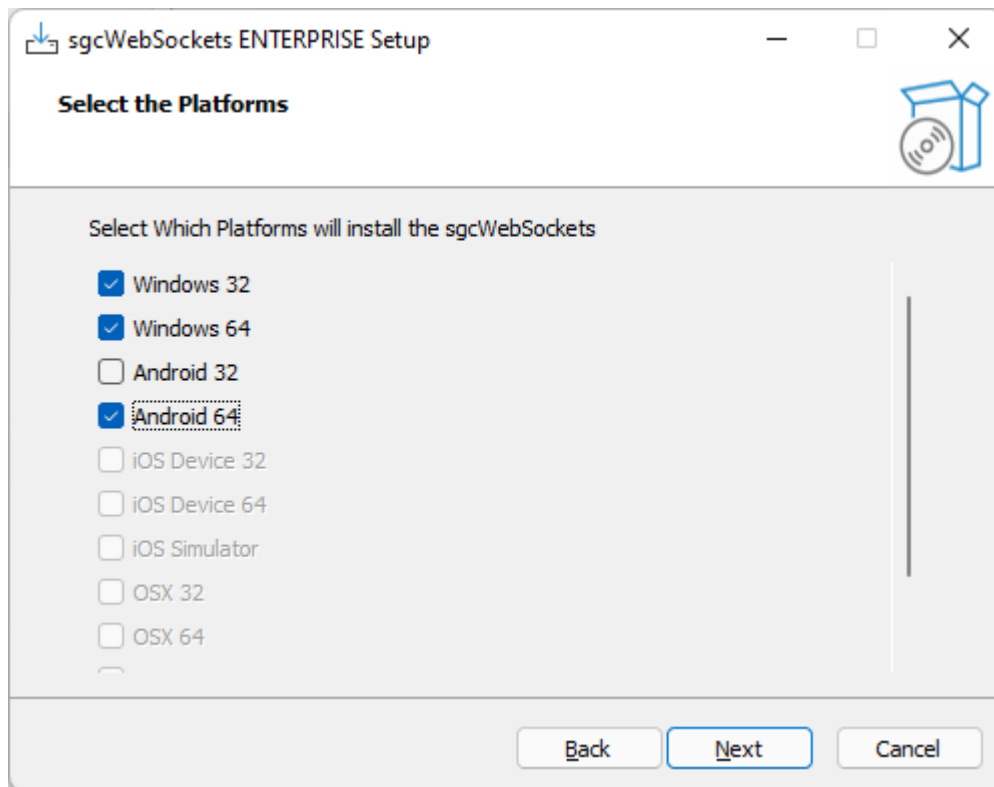
- 每次使用安装程序时可以自定义某些选项，点击 **Options** 按钮以访问这些属性。
 - **Build Packages**：若选中，安装程序将尝试构建包。
 - **Register Paths IDE**：若选中，安装程序将在 IDE 中注册所需的库路径。
 - **Register BPLs IDE**：若选中且安装程序已成功构建包，则安装程序将在 IDE 中注册设计时包。
- 以下选项仅适用于含源代码的许可证：
 - **Build Intraweb**：如果选中，安装程序将安装所需的 Intraweb 文件（默认禁用）。目前安装 Intraweb XVI 版本。
 - **Build CBuilder Dproj**：如果选中，安装程序将使用 sgcWebSockets Delphi 包构建 CBuilder 包，并生成所有必要的 CBuilder 文件。
 - **Use Custom Indy**：（仅 Enterprise 版），如果选中，sgcWebSockets 将使用自定义 Indy 版本（支持 OpenSSL 1.1 和 3.0、TLS 1.3、ALPN 等）
 - **sgcIndy Already Installed**：如果已安装 sgcIndy 包且希望使用该包编译 sgcWebSockets 包，请勾选此选项。
 - **sgcIndy Compatibility Mode**：如果 sgcIndy 包以兼容模式安装（因为其他包如 DevExpress 也在使用 Indy），请勾选此选项。
 - **Always use of the following OpenSSL API Versions**：如果要强制使用 OpenSSL 1.1.1 或 OpenSSL 3.0.0 API，请勾选此选项
 - **Debug Mode**：将调试消息保存到日志文件。请勿在生产环境中使用此模式。
 - **Build Rad Studio IDE Win64**：为 Rad Studio IDE 12+ 64 位构建设计时包。
 - **Include Resources**：默认启用。将 JavaScript 客户端包（sgcResources.RES）嵌入运行时包。仅当您的服务器通过 HTTP 向远程浏览器提供 sgcWebSockets 浏览器客户端时，才需要嵌入资源。取消勾选此选项可在编译包之前在 sgcVer.inc 中取消定义 SGC_RESOURCES，从而减小链接源代码的每个可执行文件的大小。对于 Delphi / C++Builder 纯客户端应用程序以及不提供捆绑 JS 客户端的服务器，可安全禁用。详见 [Configure Install](#) 中的底层编译器指令。



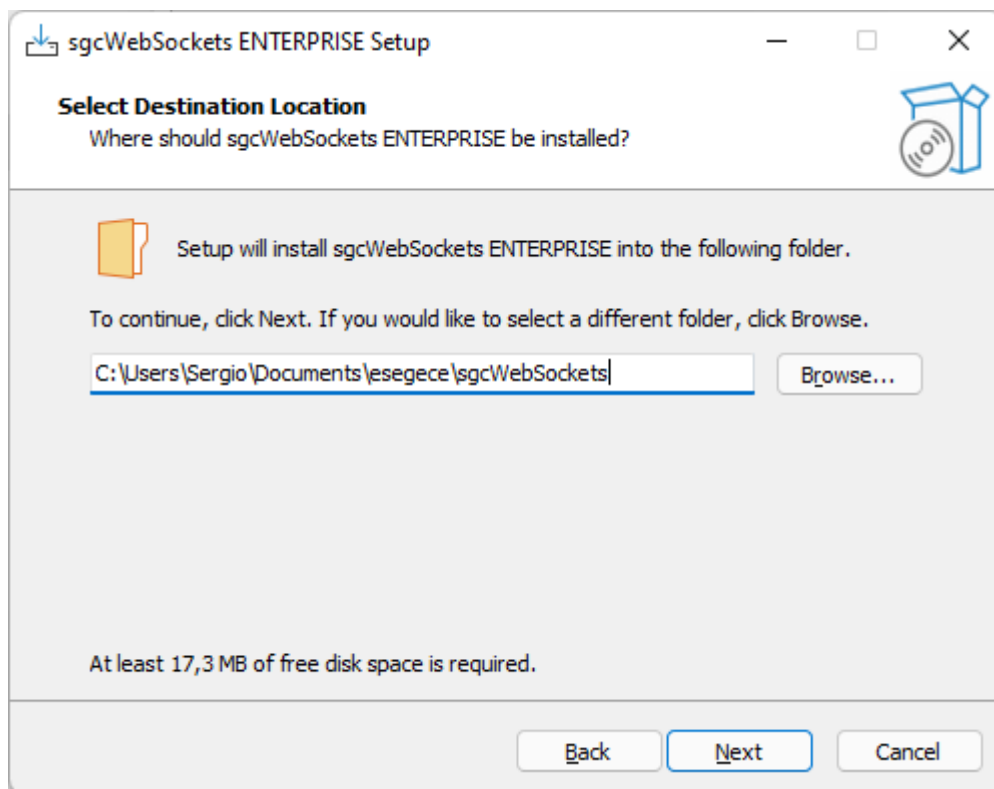
- 现在您可以选择要安装哪些 IDE 版本。只有安装程序检测到已安装的 IDE 版本才可用。



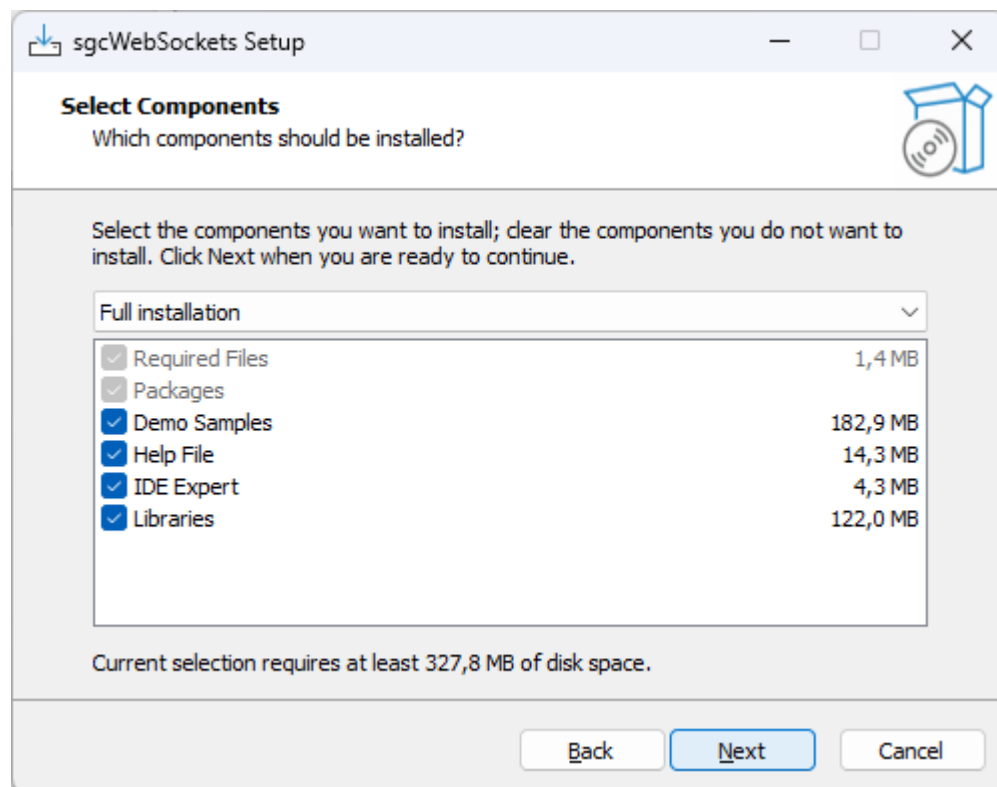
- 下一步是选择平台。



- 选择要安装软件包的文件夹。若重新安装软件包，安装程序默认会选择上次安装时所选的文件夹。



- 选择要安装的组件。已注册客户拥有一个 **IDE 专家插件**，允许您从 IDE 连接到 eSeGeCe 账户、了解可用更新、直接访问帮助台等更多功能。



- 最后，它将提取文件、编译并安装包，并在 IDE 中注册所需路径。

安装错误

- 如果 **库路径长度过长**，MsBuild 将引发错误。要解决此问题，请尝试从库路径中删除未使用的路径。MsBuild 的字符数限制为 32K。
- 错误 **F2110: 'tchar.h' 文件未找到**。此错误可能在为 CBuilder 编译时出现，解决方法是打开 CBuilder，关闭后再尝试重新安装。

安装命令行参数

安装程序支持以下命令。

/SILENT

向导和后台窗口不显示，但会显示安装进度。

/VERYSILENT

当安装程序非常静默时，不会显示此安装进度窗口。

/EXTRACT

该软件包仅被解压，并未安装。可使用 /EXTRACT=安装路径 自定义安装路径。如果您只想提取文件而不需要用户交互，请使用此参数配合 /SILENT。

/IDE

此参数允许您设置要安装的 IDE。请选择以下之一：

- delphi
- cbuilder
- radstudio

此外，您还可以添加 Lazarus。

示例：安装 Delphi 和 Lazarus。

/ide=delphi-lazarus.

/VERSIONS

使用此参数，您可以设置要安装的 RAD Studio 版本。允许多个选项：

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

使用值 "All" 可安装所有可用版本。

示例：安装 Delphi 10 和 Delphi 13。

/versions=D10-D13

/PLATFORMS

使用此参数可以设置要安装的 RAD Studio 平台。允许多个选项：

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

使用值 "All" 安装所有可能的平台。

示例：安装 Win32 和 Win64。

/platforms=Win32-Win64

/USERNAME

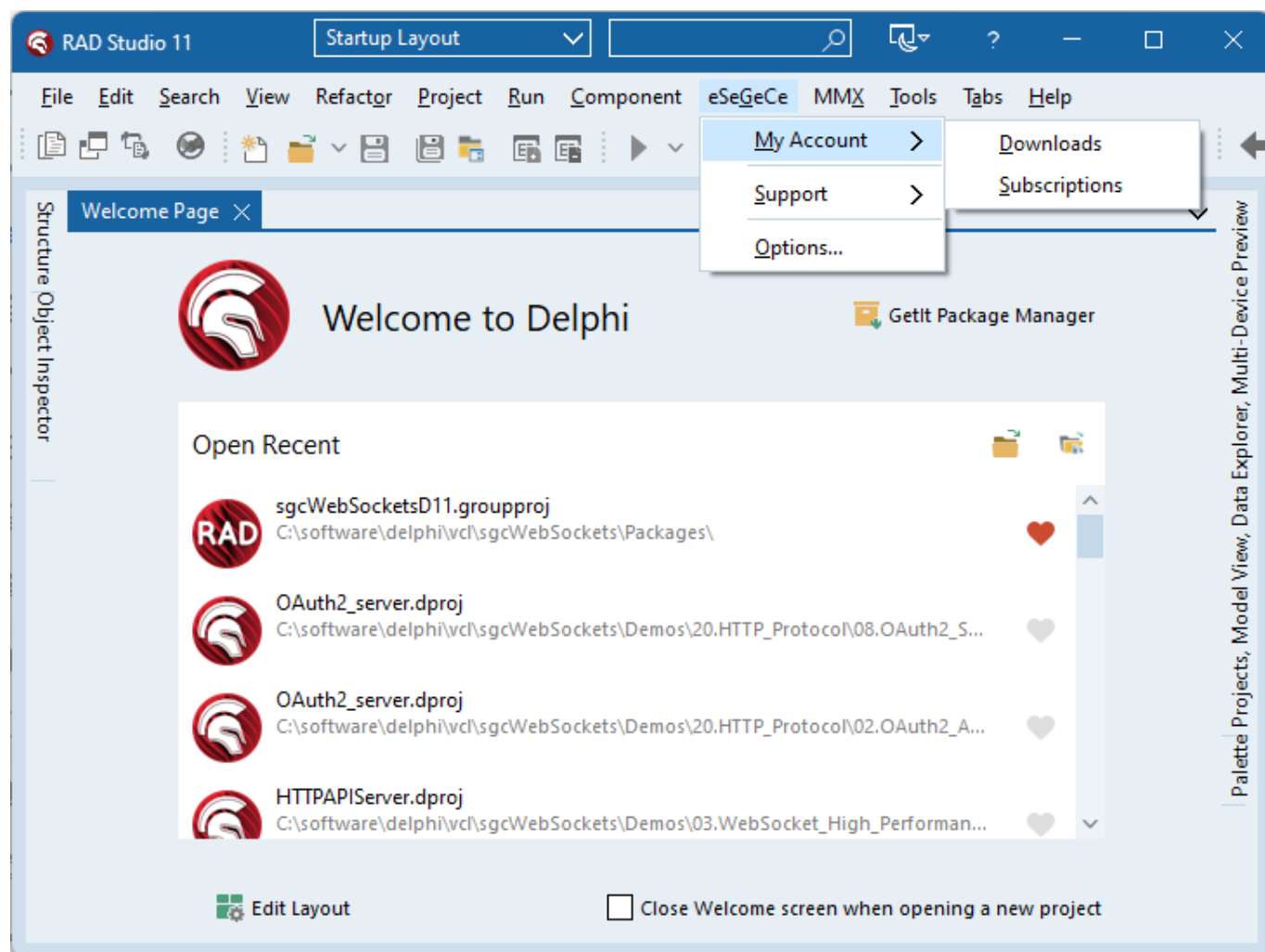
设置订阅的用户名

/PASSWORD

设置订阅的密码

IDE 专家

如果已安装 IDE Expert，您将看到以下菜单选项：

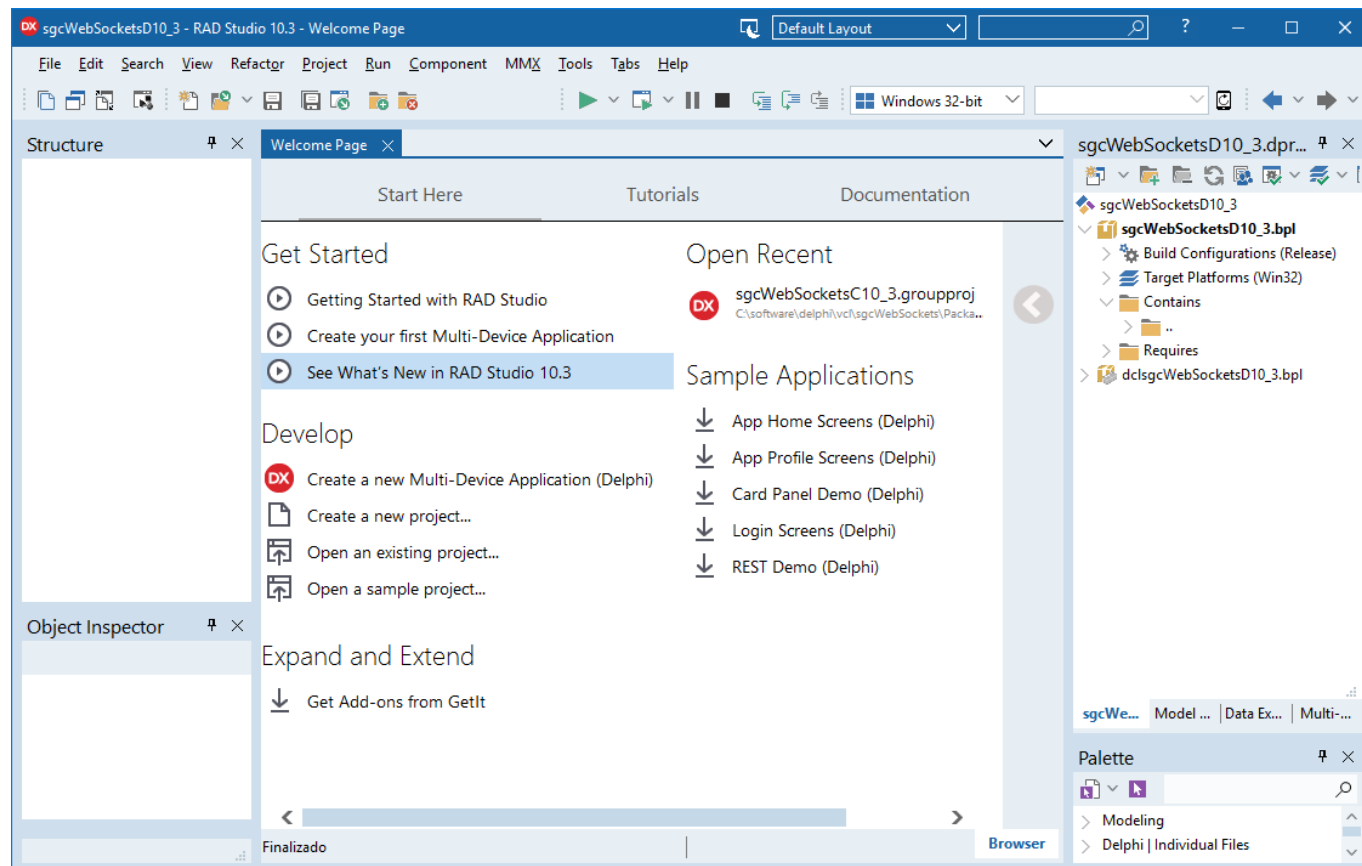


- **MyAccount**：直接访问"下载"菜单（您可以在这里下载最新 beta 版本）以及"订阅"菜单，以管理您的许可证或续期已过期的许可证。
- **支持**：直接访问帮助台或论坛并自动登录。文档和联系表单也可访问。
- **选项**：在此菜单中，您可以配置账户的用户名/密码。选择默认浏览器并检查是否有可用更新。

手动安装包

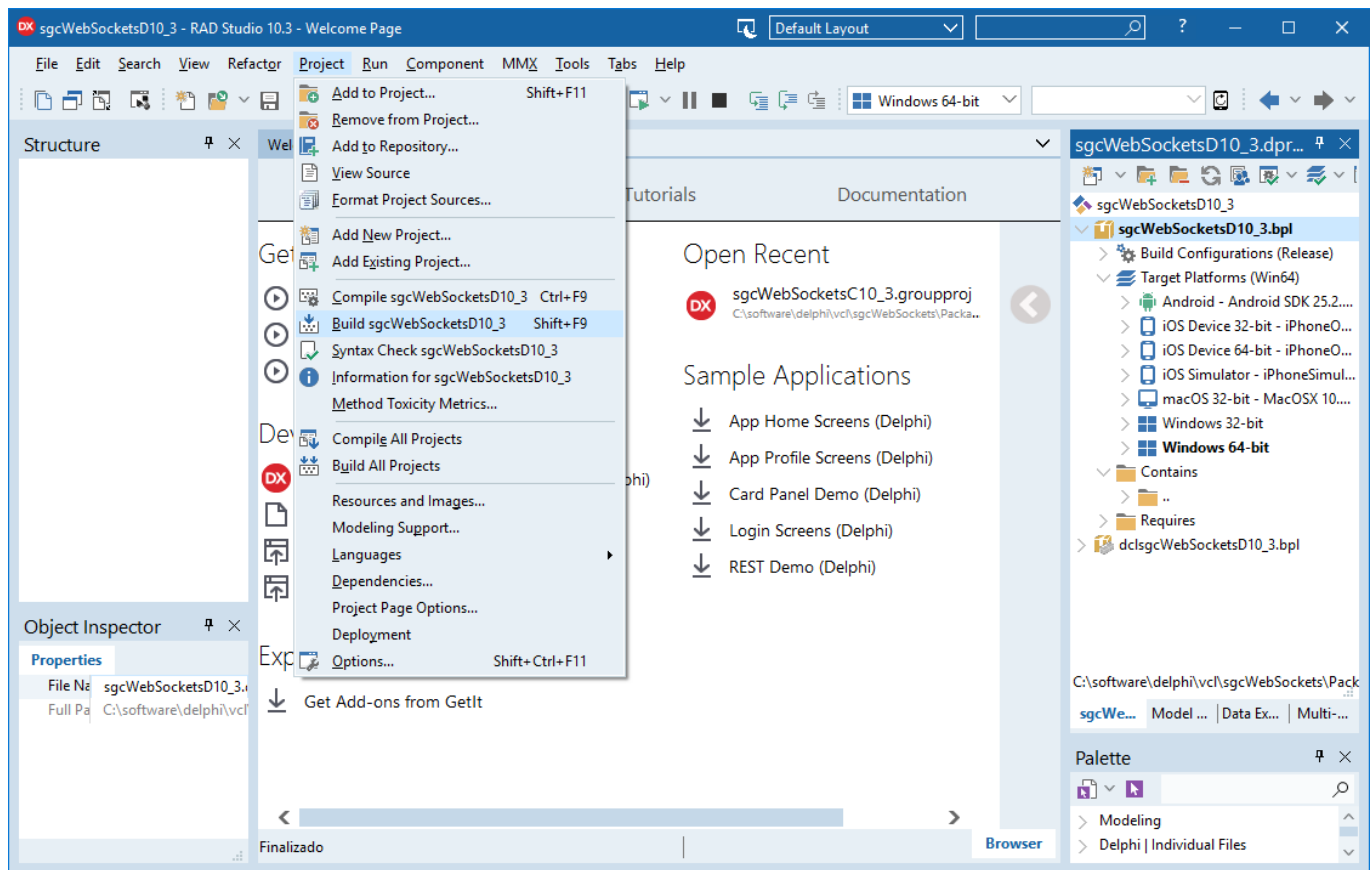
按照以下步骤安装 sgcWebSockets 包，截图使用的是 Delphi 10.3 版本。

1. 打开 sgcWebSocketsD10_3 项目组。

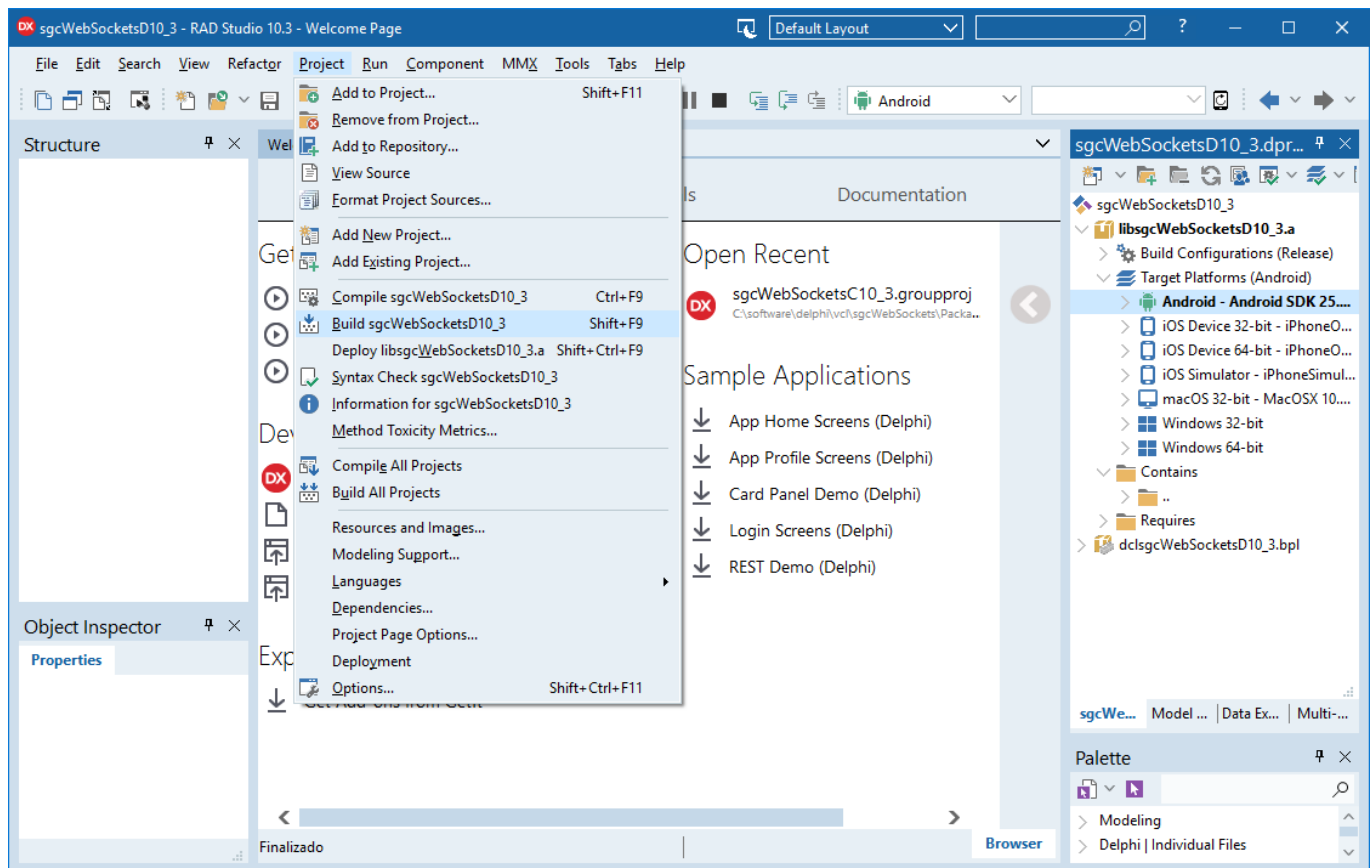


2. 现在必须**首先编译运行时包**（名称以 sgcWebSockets 开头）。每个目标平台对应一个包，且依赖于 Delphi 版本，因此请**逐一选择目标平台并构建每个包**。

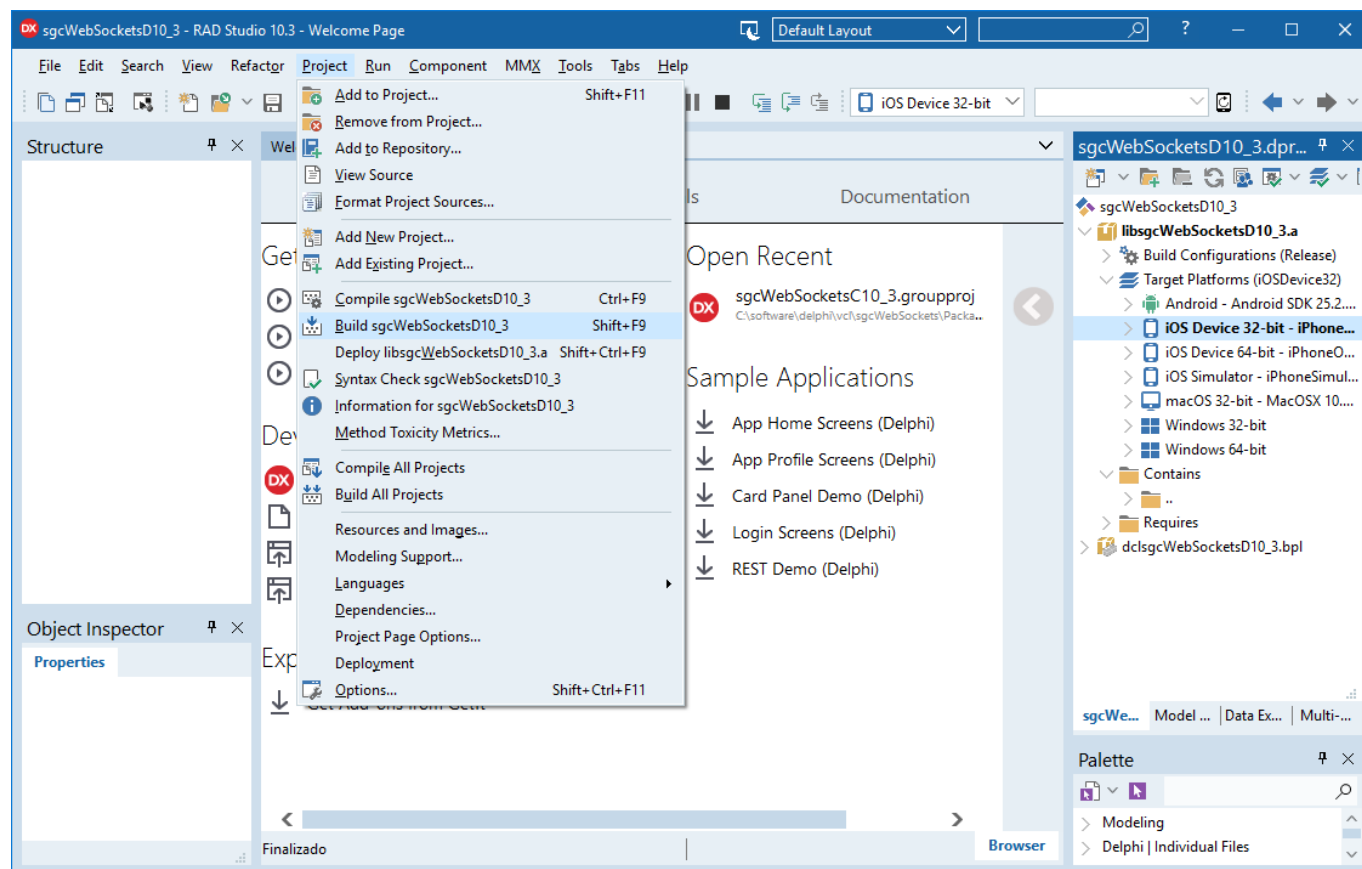
3. 选择 win64 作为目标平台并构建包。



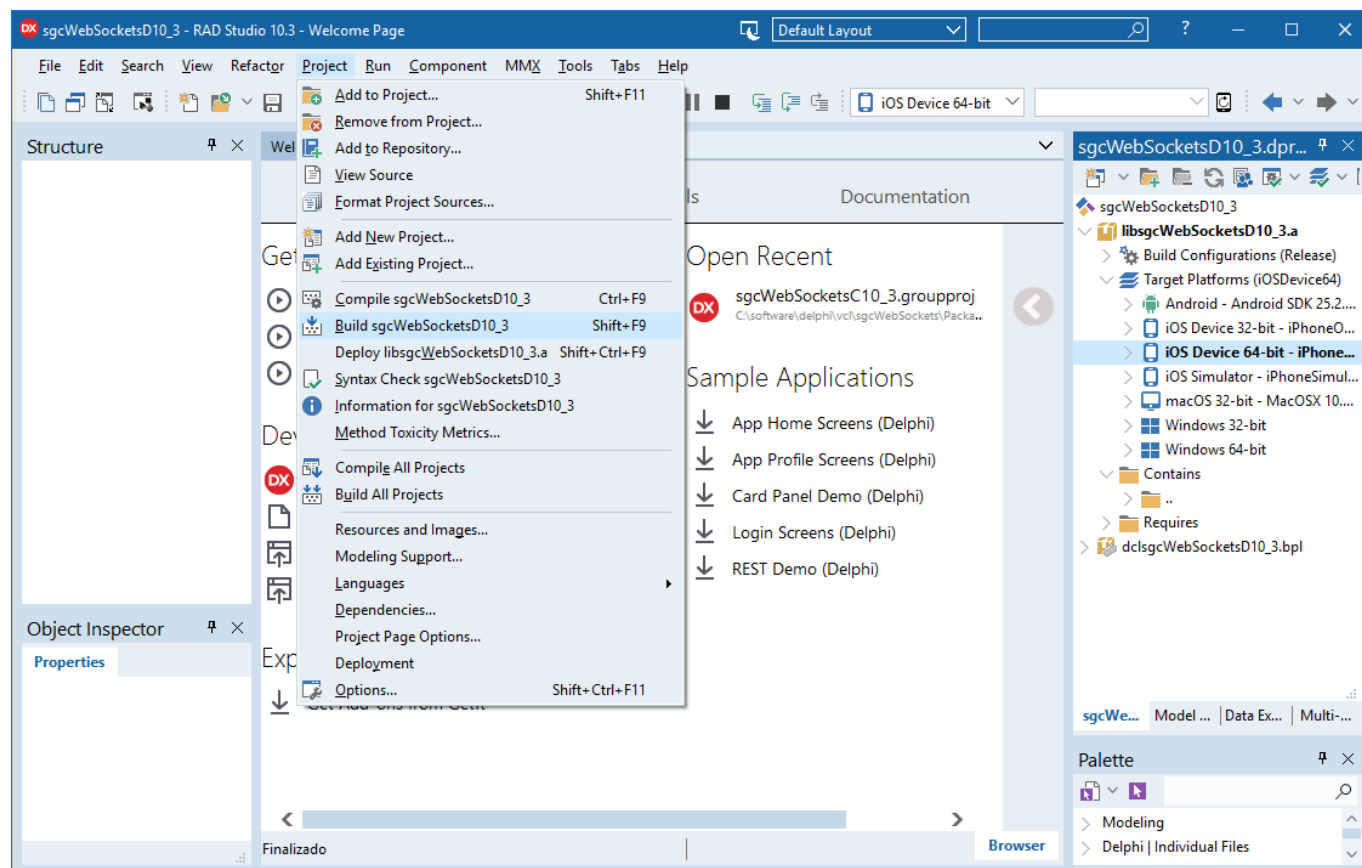
4. 选择 Android 作为目标平台并构建软件包。



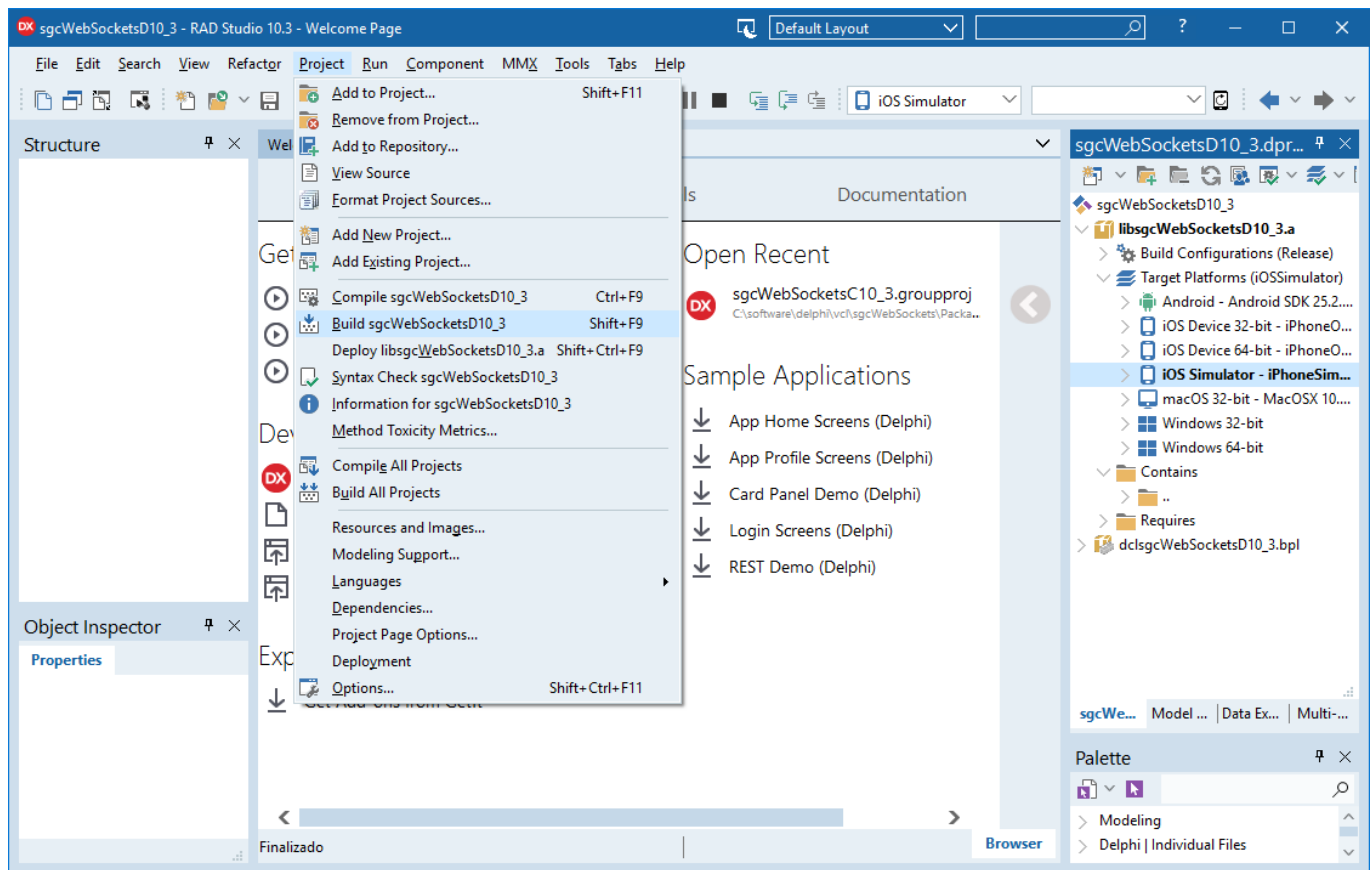
5. 选择 iOS Device 32 作为目标平台并构建包。



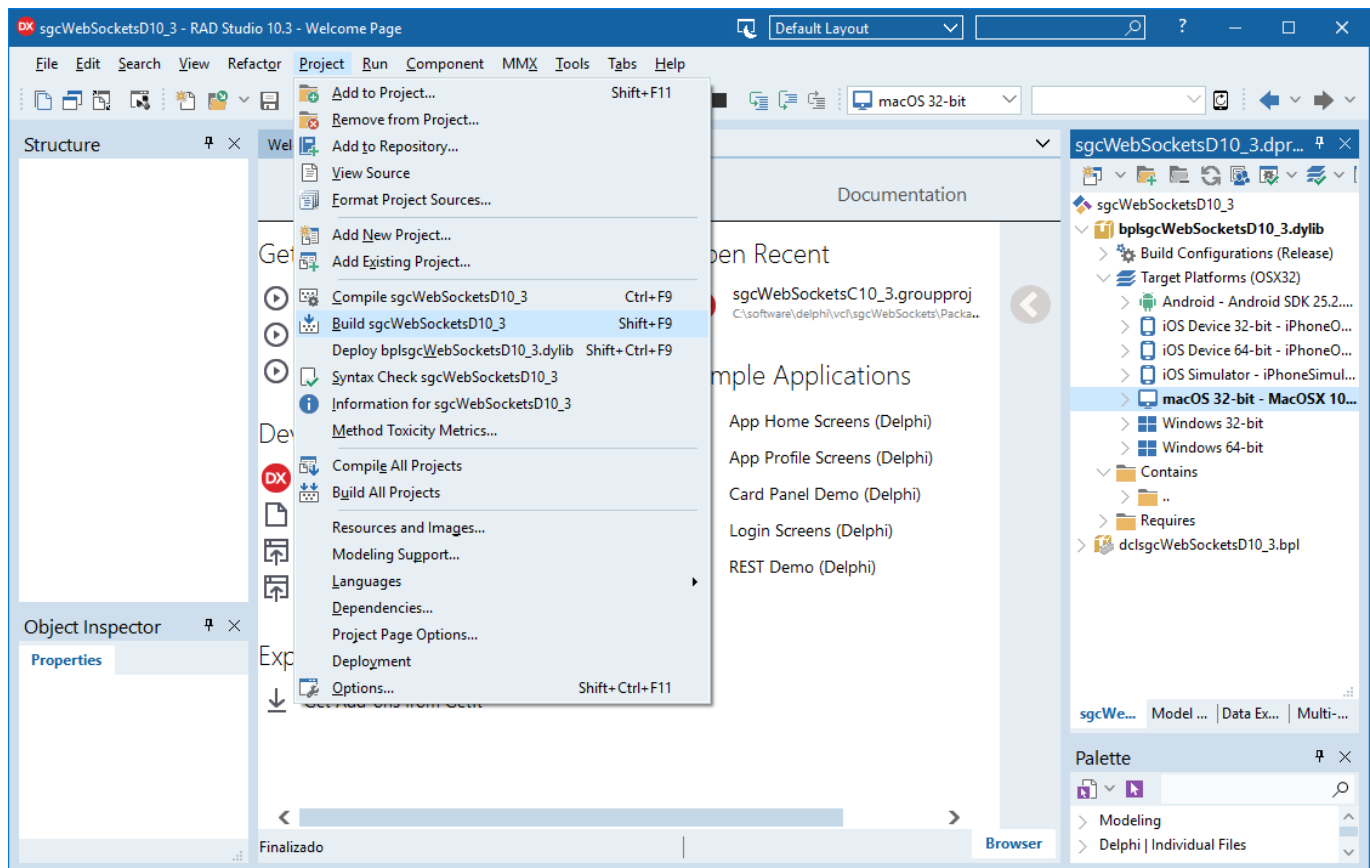
6. 选择 **iOS Device 64** 作为目标平台并构建包。



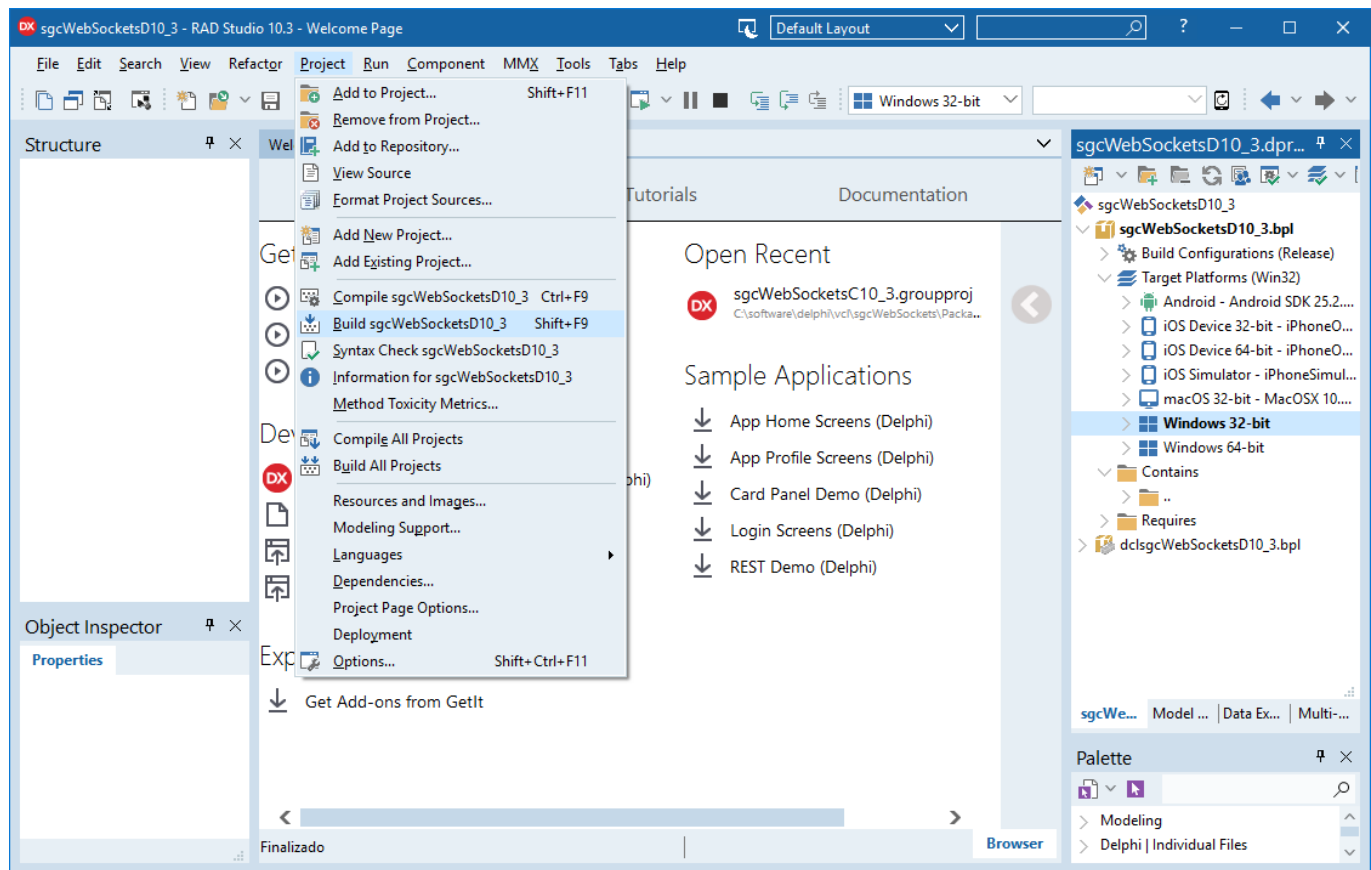
7. 选择 **iOS Device Simulator** 作为目标平台并构建包。



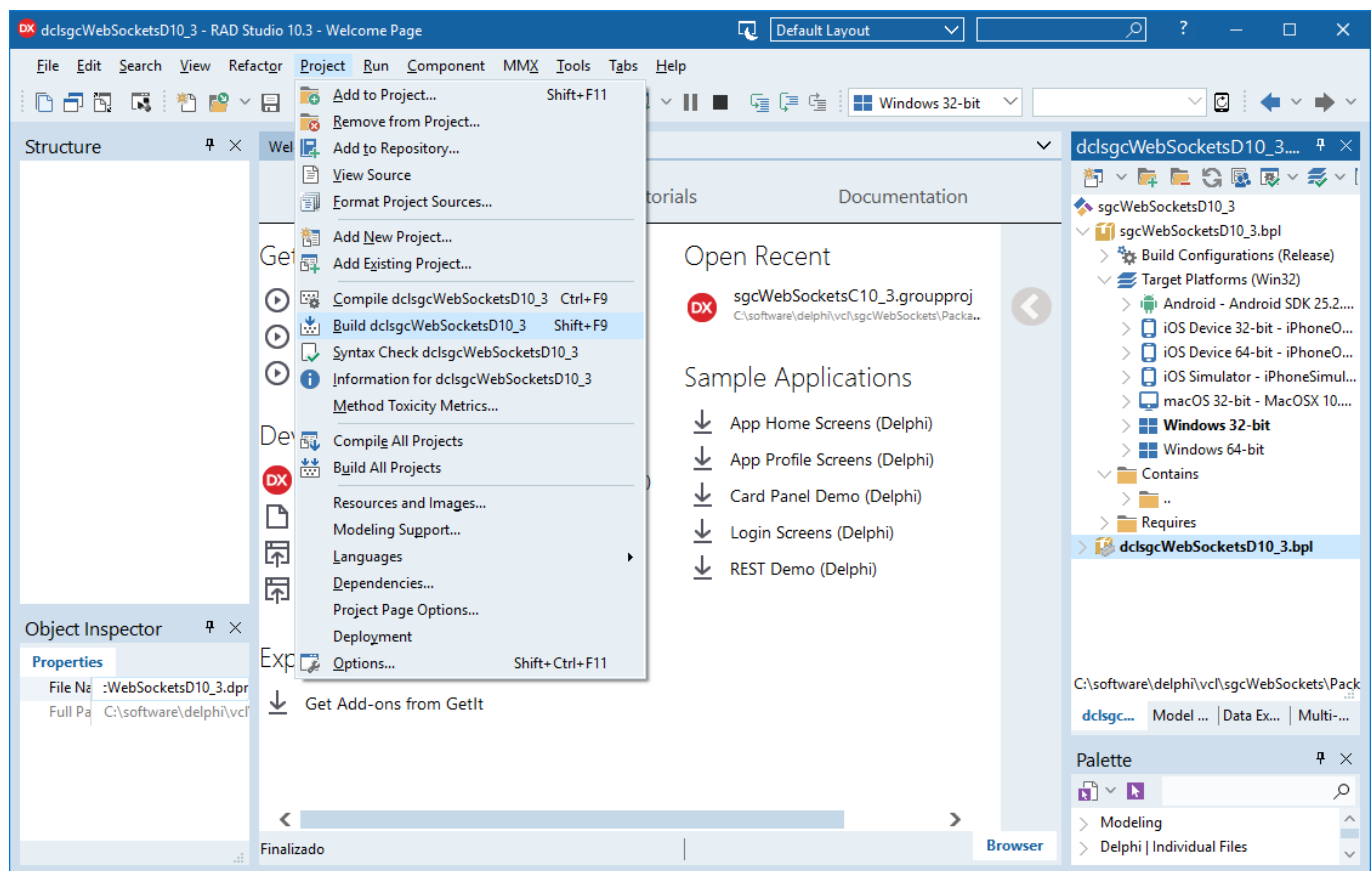
8. 选择 **MacOS 32** 作为目标平台并构建软件包。

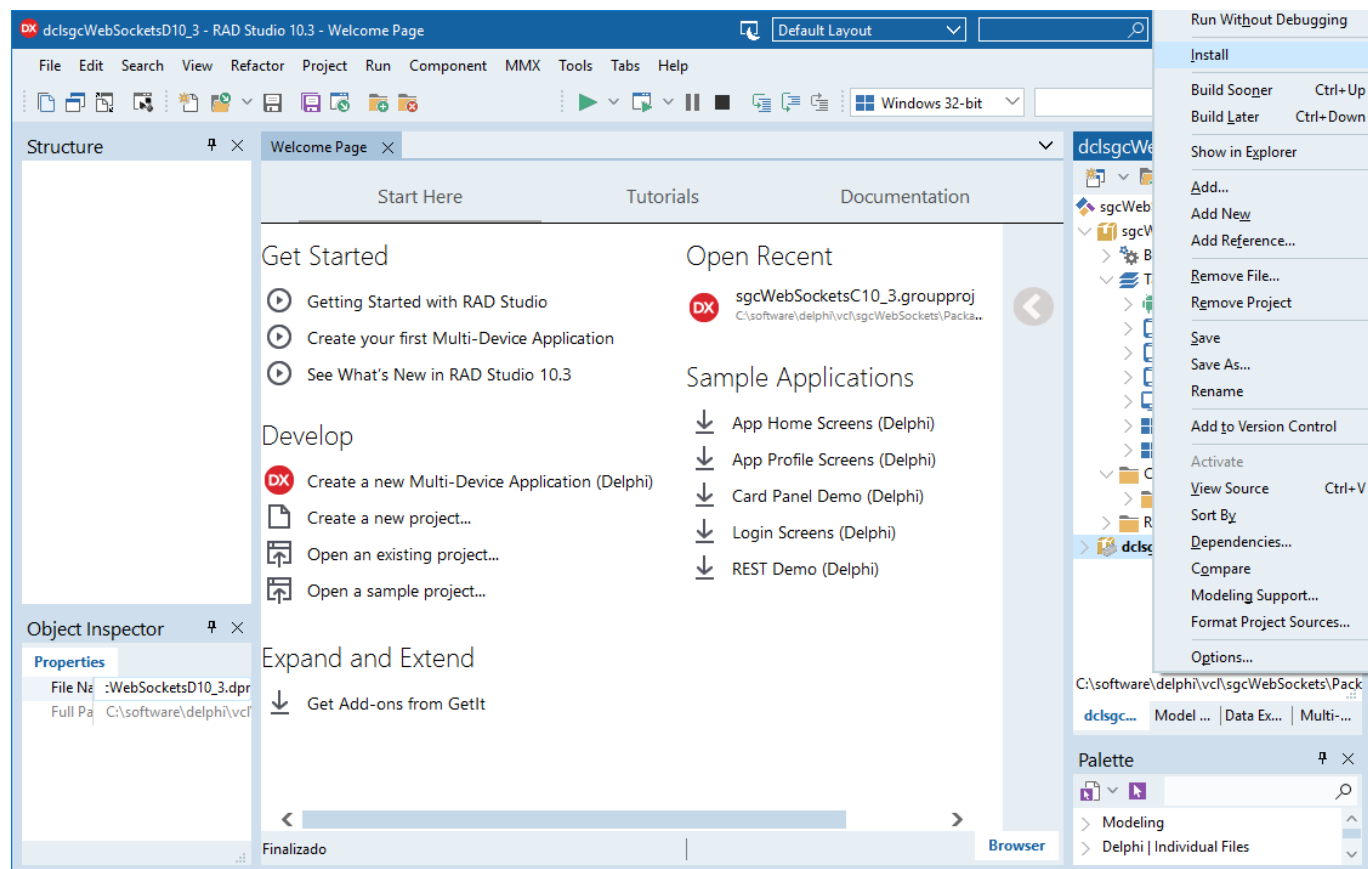


9. 选择 **Win32** 作为目标平台并构建软件包。

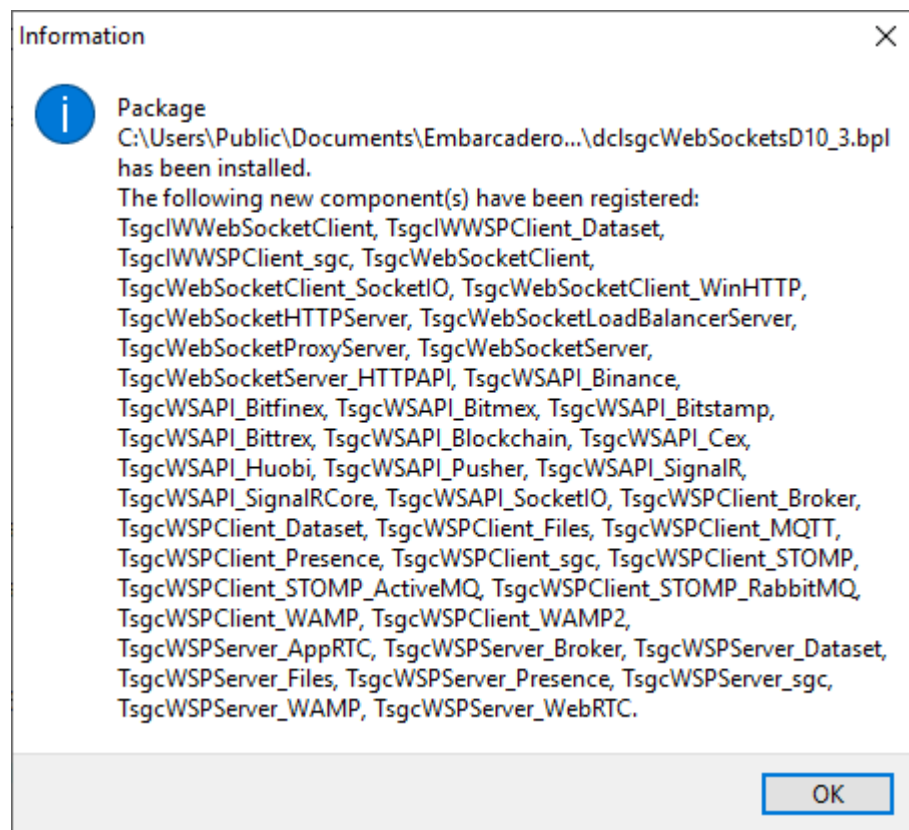


10. 所有运行时包编译完成后，选择设计时包（名称以 dcl 开头），先构建然后安装（设计时包仅以 Win32 为目标平台）。

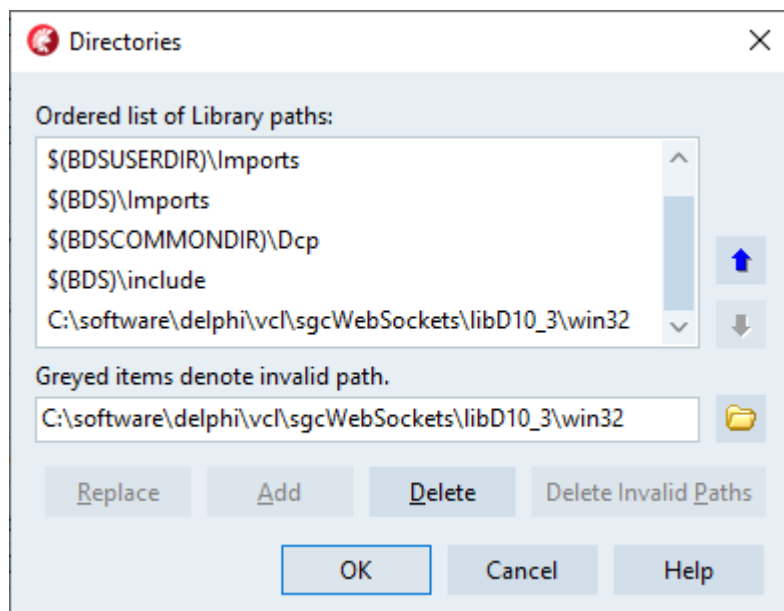




11. 如果安装成功，您将看到一条包含所有已安装组件的消息。



12. 然后，您只需将编译文件所在目录添加到您的 **Rad Studio Library Path** 中。每个目标平台（win32、win64、osx64.....）都必须添加。



* 如果您使用的是 **Datasnap** 服务器，这些服务器**未包含**在 sgcWebSockets 包中，因为它们无法安装；它们是仅运行时组件。在这种情况下，您必须将 Source 文件夹也添加到库路径中。

安装错误

安装组件时，您有时可能会遇到一些错误。

找不到 Intraweb 包

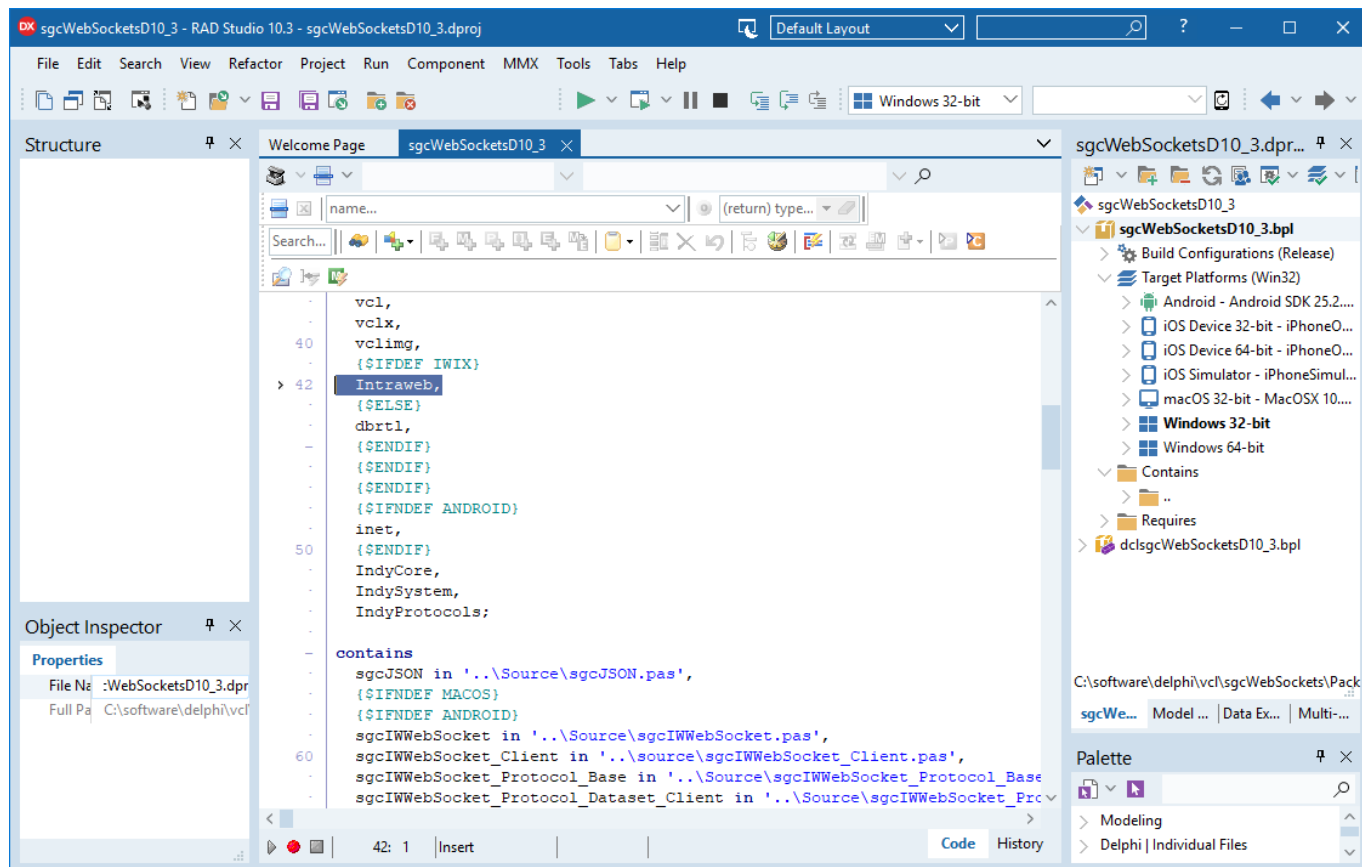
sgcWebSockets 使用 Delphi 附带的默认 **Intraweb** 版本编译。如果您未安装 Intraweb，可以修改位于 Source 文件夹中的 **sgcVer.inc** 文件。

搜索您的 Delphi 版本，并注释掉所有 **Intraweb** 的编译器定义（以 IW 开头）。示例：对于 Delphi 10.4，注释掉所有 Intraweb 的编译器定义

```
{ $IFDEF VER340 } { Delphi 10.4 }
{$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}

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

如果已安装 Intraweb 但其版本与 Delphi 默认附带的版本不同，您的 Intraweb 包可能具有不同的名称。此时请打开 sgcWebSockets 运行时包，并在项目源代码中**更改 Intraweb 名称**。



未找到 Indy 包

sgcWebSockets 需要 **Indy** 才能在您的 IDE 中安装组件。Trial 安装程序针对 Delphi / CBuilder 附带的 Indy 库编译，因此如果您收到如下消息：

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

您很可能拥有较新版本的 **Indy**，因此要安装试用版，必须删除此版本并使用 Delphi / CBuilder 安装程序安装内置的 Indy 版本。

如果您拥有完整源代码，只需检查以下内容：

1. 所需的 Indy 包：IndyCore、IndySystem 和 IndyProtocols。如果您有较新的 Indy 版本，这些包的名称可能不同，请访问菜单 **"Component / Install Packages"**，检查 Indy 包的名称，并在项目中相应修改。
2. sgcWebSockets 支持多个 Indy 版本，有编译器定义允许为每个 Indy 版本编译。打开位于源文件夹的 **sgcVer.inc**，根据您的 Indy 版本进行相应更改（是 IdVers.inc Indy 文件的 gslVersion）。部分编译器定义：

```

{$DEFINE INDY10_1}
{$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 在 CBuilder 中

如果使用 CBuilder 编译项目时出现此错误，请在项目中设置以下选项：

Project > Options > C++ Linker
取消勾选"与动态 RTL 链接"

Project > Options > Packages > Runtime Packages
取消勾选"与运行时包链接"

在 CBuilder Win64 中找不到包导入：sgcWebSocketsCXXX.bpi

编译 win64 运行时软件包时，必须同时编译 Release 和 Debug 版本。

System.ZLib.hpp 和 IdZLib.hpp 引用不明确 CBuilder

sgcWebSockets Standard 和 Professional 对某些组件使用 Indy，而 Indy 不使用 ZLib 单元，而是使用自己的 ZLib 副本。项目同时链接到 ZLib 和 Indy ZLib 单元，因此编译时编译器不知道哪个是正确的引用。有两种解决方案：

1. 搜索包含 System.ZLib.hpp 链接的位置，并将其删除或移至 IdZLibHeaders.hpp 之后
2. 在 项 目 选 项 中 使 用 以 下 条 件 定 义 NO_USING_NAMESPACE_SYSTEM_ZLIB 或 DELPHIHEADER_NO_IMPLICIT_NAMESPACE_USE，以避免使用 System.Zlib.hpp。

歧义引用 System.ZLib.hpp 和 sgclIdZLib.hpp CBuilder

sgcWebSockets Enterprise 对部分组件使用自定义 Indy 版本，而 Indy 不使用 ZLib 单元，而是使用其自己的 ZLib 副本：sgclIdZLib、sgclIdZLibHeaders.....项目链接了 ZLib 和 Indy ZLib 单元，因此编译时编译器无法确定正确的引用，因为名称相同。有两种解决方案：

1. 搜索包含 System.ZLib.hpp 链接的位置，并将其删除或移至 sgclIdZLibHeaders.hpp 之后。
2. 在 项 目 选 项 中 使 用 以 下 条 件 定 义 NO_USING_NAMESPACE_SYSTEM_ZLIB 或 DELPHIHEADER_NO_IMPLICIT_NAMESPACE_USE，以避免使用 System.Zlib.hpp。

CBuilder 和 Android 上出现 Sgcwebsocket... 的 vTable 未定义引用

使用以下解决方法修复该错误：通过菜单"Project/Add to Project"将位于 dcp/android 默认文件夹中的 libsgcwebsocketsC*.a 文件添加到您的项目。

示例：对于 CBuilder 11，请将文件 "libsgcWebSocketsC11.a" 添加到您的项目中，该文件默认位于文件夹 "C:\Users\Public\Documents\Embarcadero\Studio\22.0\DCP\Android\Release"。

在 Lazarus 下校验和发生变化

如果在 Lazarus 下安装组件时构建 IDE 的配置文件不是"Optimized IDE"，则可能引发此错误。试用版是使用"Optimized IDE"配置文件编译的。

找不到 Y 使用的 X，ppu 不兼容

请尝试以下解决方法：从菜单选项中选择"运行 / 清理并重新构建"。

配置安装

在源代码文件夹中，有一个名为 `sgcVer.inc` 的文件，其中包含所有 Delphi、CBuilder 和 Lazarus IDE 的编译器定义。

在此处您可以自定义 Intraweb、Indy 等的配置，**通常无需进行任何更改**，除非您想启用/禁用某些功能。

请谨慎更改编译器定义，如需帮助请联系我们。

每个 Delphi 版本都有一个专区，您可以在其中配置所有编译器定义，以下是 Delphi 10.4 的示例。

```
{$IFDEF VER340} { Delphi 10.4 }
{$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}

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

Indy

Indy 库有一些编译器定义，具体取决于已安装的 Indy 版本，默认配置为 Delphi 附带的 Indy 包。Indy 版本是 IdVers.inc Indy 文件的 gsidVersion 参数。

Intraweb

若未安装 IntraWeb，只需注释掉 IntraWeb 的编译器定义（以 IW... 开头的定义）即可。

SGC_RESOURCES

默认情况下，sgcVer.inc 声明 `{ $DEFINE SGC_RESOURCES }`，这会导致运行时包嵌入资源文件 `sgcResources.RES`。这会为每个静态链接 sgcWebSockets 源码的可执行文件增加额外字节。

该资源包含捆绑的 JavaScript 客户端代码，sgcWebSockets 服务器可通过 HTTP 将其传递给远程浏览器（从而使浏览器能够使用 sgc 子协议）。仅在以下情况下才需要：

- 您使用 sgcWebSockets HTTP/WebSocket 服务器向浏览器提供 JS 客户端包（典型用于基于 TsgcWSProtocol_JS_* 的服务器）。

以下情况无需：

- Delphi / C++Builder WebSocket 客户端（数据在原生代码中消费，而非在浏览器中）。
- 不提供捆绑 JS 客户端的服务器（例如仅 WebSocket 服务器、原生对原生服务器）。

要排除嵌入资源，请在 sgcVer.inc 中注释掉相应行：

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

或者，运行源码版 [安装程序](#) 时，取消勾选选项页面上的 **包含资源** 选项；安装程序将在编译包之前重写 sgcVer.inc。

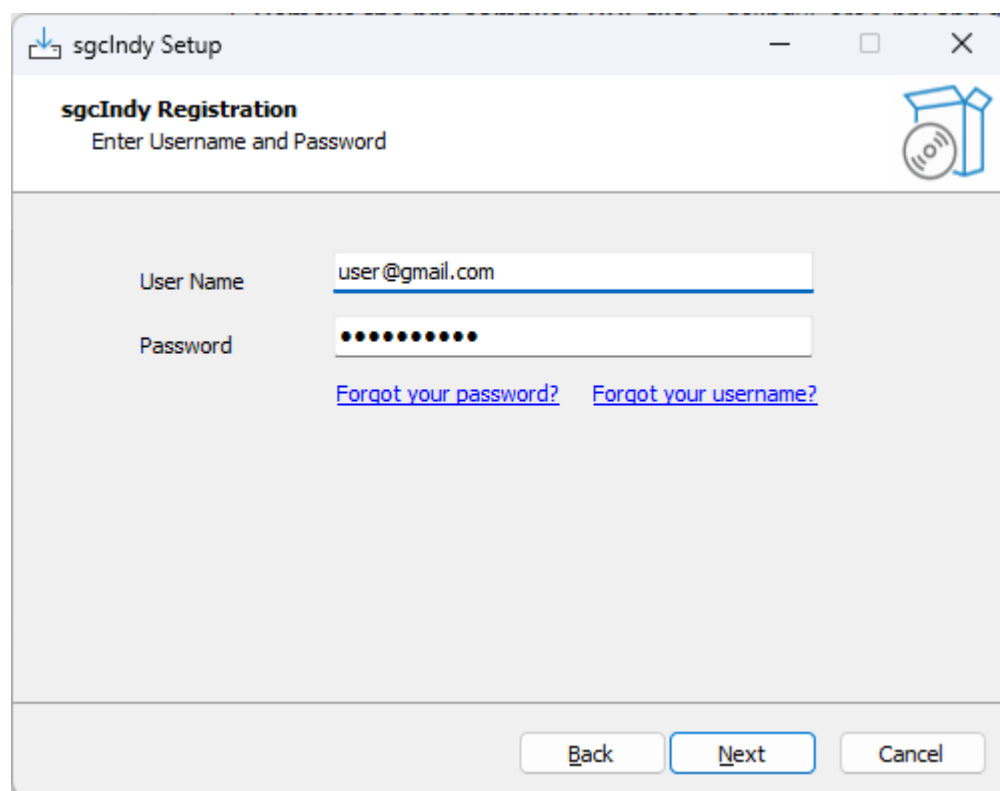
安装 sgclndy 包

安装配置

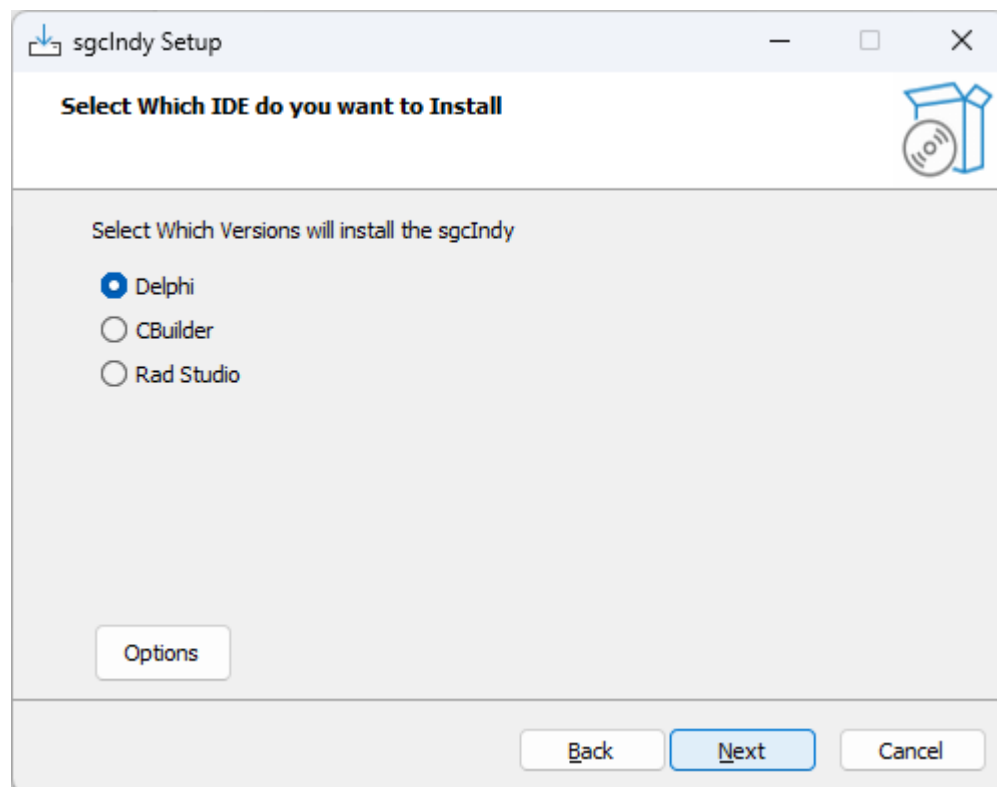
*最低要求 Windows Vista（不支持 Windows 2000、XP 和 Server 2003）。

购买了许可证的用户可以使用安装程序安装 sgclndy 软件包。以下是安装软件包的逐步说明。

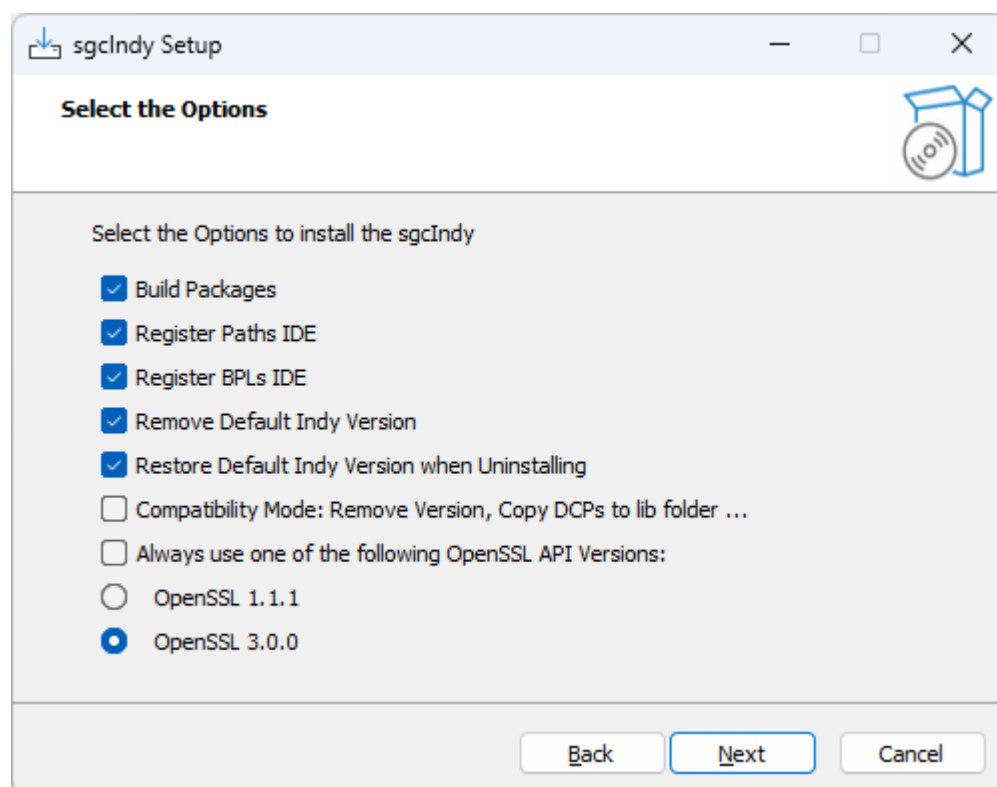
- 执行安装程序。
- 首先，您必须设置您的 eSeGeCe 私有账号的用户名/密码。此操作只需输入一次，下次使用安装程序时，安装程序将读取最新值。

The image shows a Windows-style dialog box titled "sgclndy Setup". Inside the dialog, the text "sgclndy Registration" is displayed, followed by the instruction "Enter Username and Password". There is a small icon of a box with a coin on the right side. Below the text, there are two input fields: "User Name" with the text "user@gmail.com" entered, and "Password" with a series of dots. Below the password field, there are two blue links: "Forqot your password?" and "Forqot your username?". At the bottom of the dialog, there are three buttons: "Back", "Next", and "Cancel". The "Next" button is highlighted with a blue border.

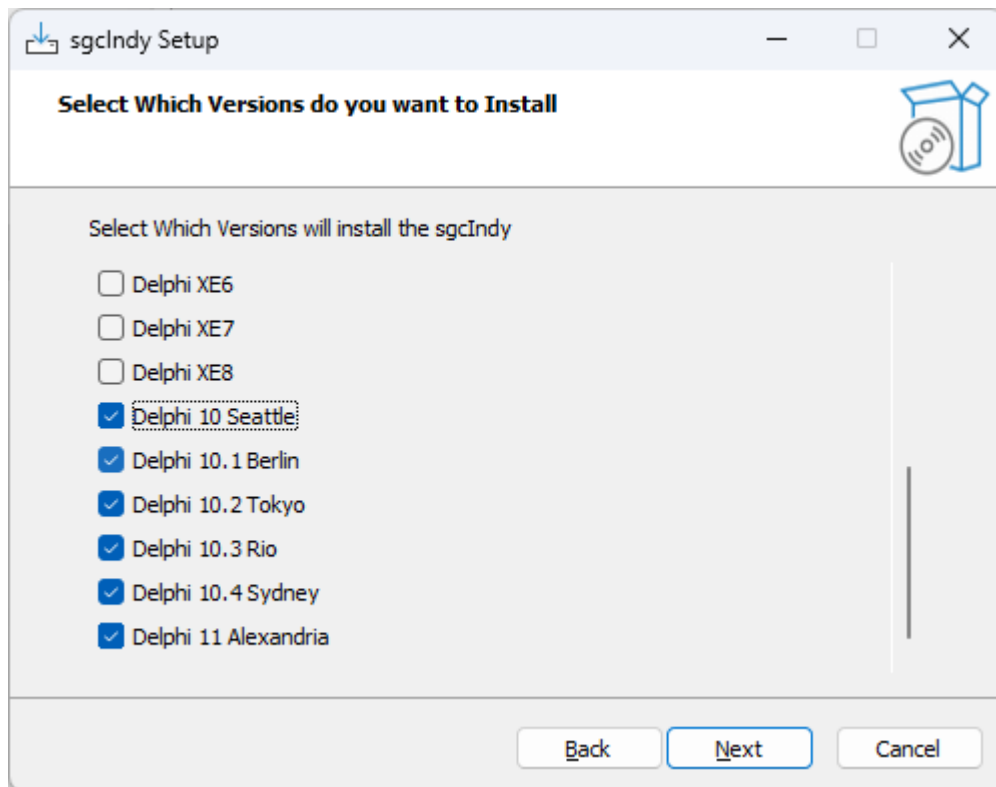
- 若用户已成功登录，请选择要在 Delphi、CBuilder 还是 Rad Studio IDE 中安装。



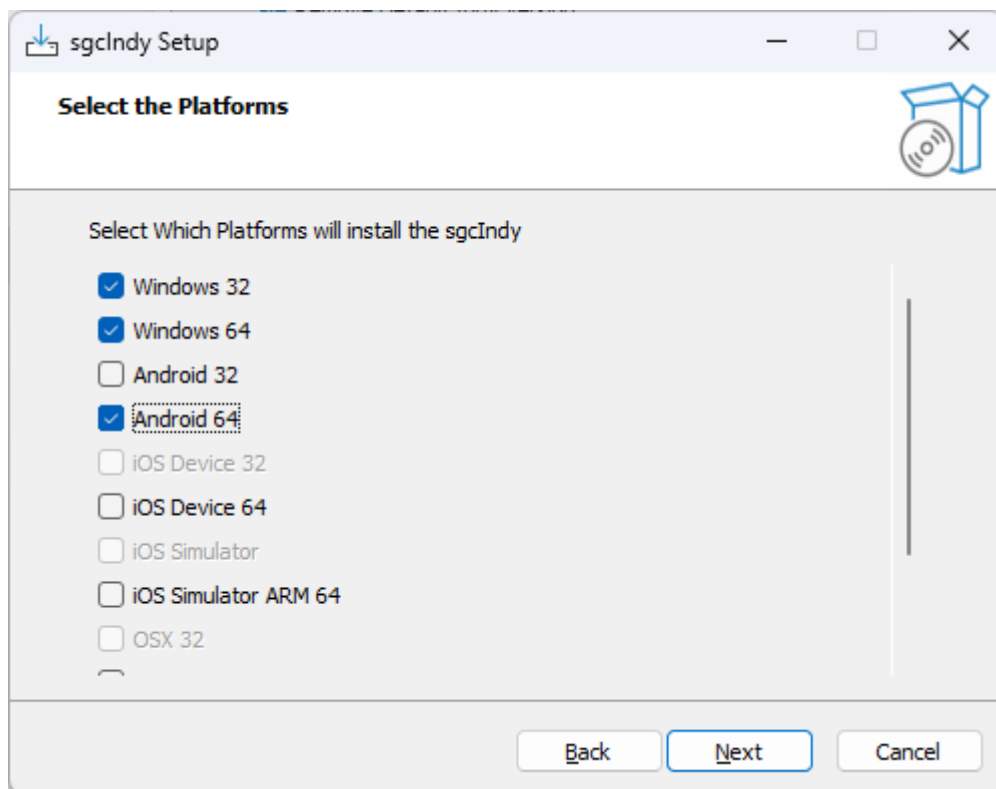
- 每次使用安装程序时都有一些可以自定义的选项，点击 **Options** 按钮访问这些属性。
 - **Build Packages**：如果选中，安装程序将尝试构建包。
 - **Register Paths IDE**：如果选中，安装程序将在 IDE 中注册所需的库路径。
 - **Register BPLs IDE**：如果选中且安装程序已成功构建包，安装程序将在 IDE 中注册设计时包。
 - **Remove Default Indy Version**：如果选中，安装程序将首先卸载 Rad Studio 附带的标准 Indy 版本。
 - **Restore Default Indy Version when Uninstalling**：如果选中，卸载包时安装程序将回滚已卸载的标准 Indy 版本。
 - **Compatibility Mode**：如果选中，dcp 文件将在不带版本号的情况下编译，并复制到 Embarcadero/lib 文件夹。如果其他包（如 DevExpress）使用了 Indy 包，请勾选此选项。此外，它还会使用已安装的 sgcIndy 版本编译 Embarcadero IP 抽象单元。
 - **Always use of the following OpenSSL API Versions**：如果您希望强制使用 OpenSSL 1.1.1 或 OpenSSL 3.0.0 API，请勾选此选项。



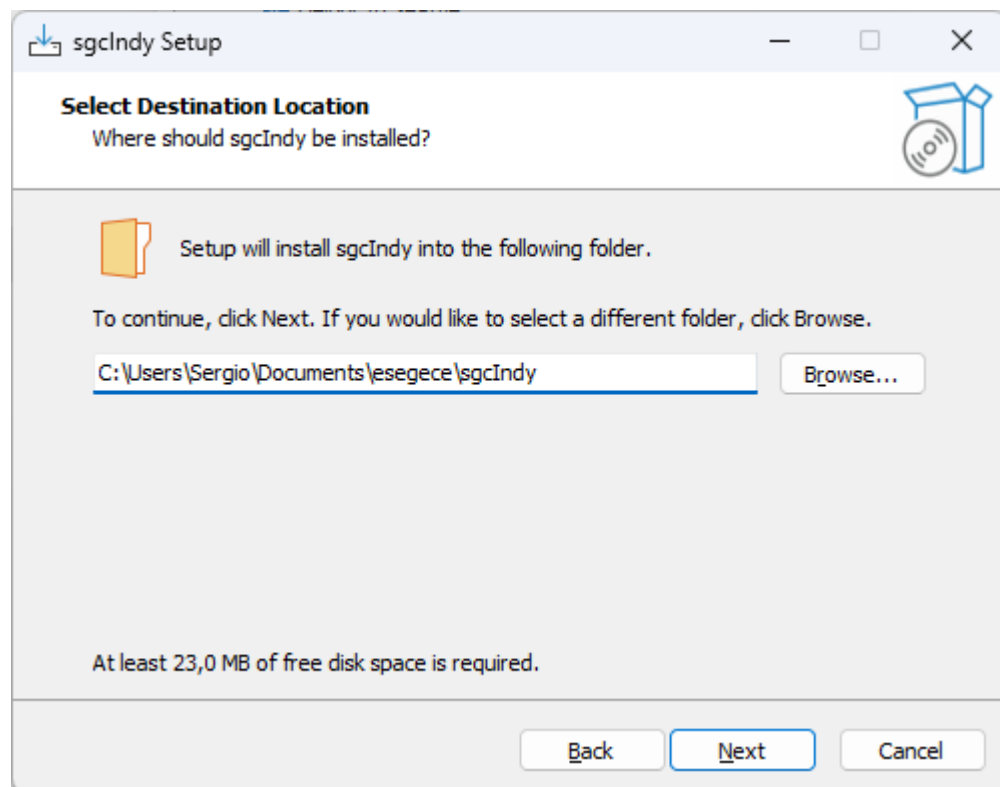
- 现在，您可以选择要安装的 IDE 版本。只有安装程序检测到已安装的 IDE 版本才会可选。



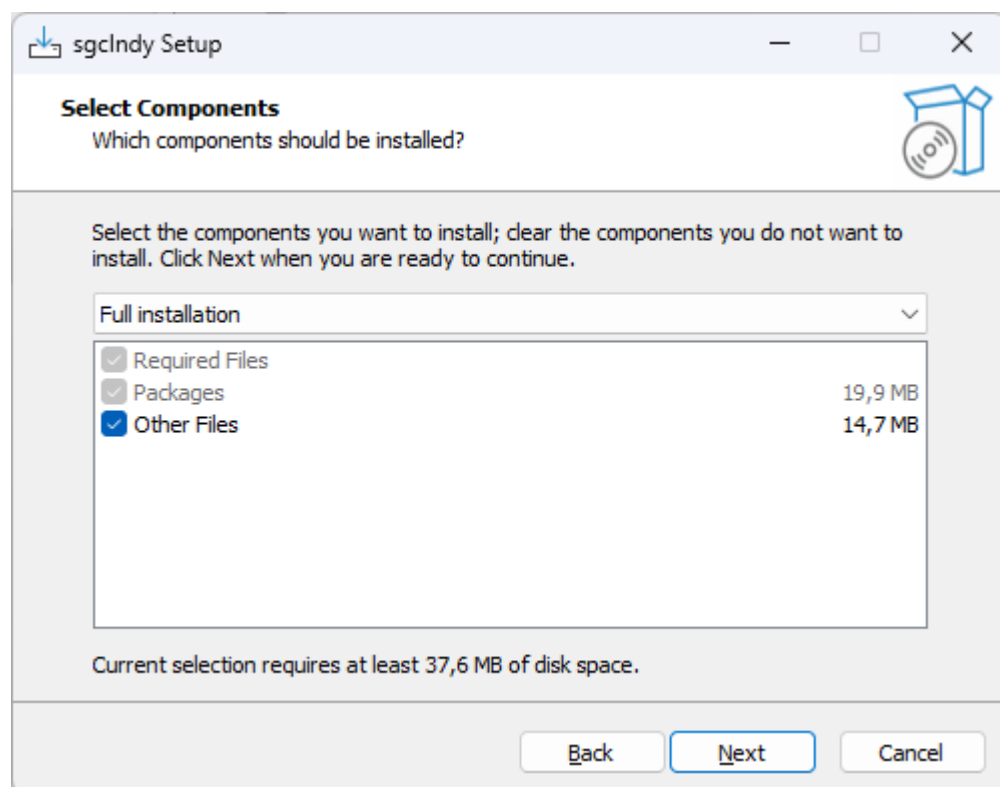
- 下一步是选择平台。



- 选择要安装软件包的文件夹。若重新安装软件包，安装程序默认会选择上次安装时所选的文件夹。



- 选择要安装的组件。



- 最后，它将提取文件、编译并安装包，并在 IDE 中注册所需路径。

安装错误

- 如果 库路径长度过长，MsBuild 将引发错误。要解决此问题，请尝试从库路径中删除未使用的路径。MsBuild 的字符数限制为 32K。

手动安装

若 Indy 已安装，则需要先卸载。

1. 通过“组件 > 安装包”对话框，从 IDE 中移除预编译的 BPL 文件——dclIndyCoreX.bpl 和 dclIndyProtocolsX.bpl。
2. 然后删除所有现有二进制文件（IndySystemX.、IndyCoreX.、IndyProtocolsX.、dclIndyCoreX. 和 dclIndyProtocolsX.*），以及删除任何 Indy 10 源文件（若存在）。
3. 请务必在 IDE 的 \bin、\lib 和 \source 文件夹、\Indy 子文件夹及操作系统系统文件夹中检查文件。

要构建 sgclIndy 软件包，您可以

1. （仅限 CBuilder）使用与您的 CBuilder 版本对应的命令行 FULLC#.BAT 脚本。
2. 在 IDE 中打开各个 DPK 文件并按以下顺序编译：
 1. IndySystemX.dpk（位于 Lib\System）
 2. IndyCoreX.dpk（位于 Lib\Core）
 3. IndyProtocolsX.dpk（位于 Lib\Protocols）
 4. dclIndyCoreX.dpk（位于 Lib\Core）
 5. dclIndyProtocolsX.dpk（位于 Lib\Protocols）

Indy 包构建完成后，前往菜单 **Components / Install Packages**，安装 Indy 设计时包。

1. dclIndyCore*.bpl
2. dclIndyProtocols*.bpl

最后，在您的 IDE 中设置 sgclIndy 包的路径。

配置 ZLib

ZLib 版本：1.2.12

sgcWebSockets 在启用 WebSocket [压缩 PerMessage Deflate](#) 扩展时使用 ZLib 压缩。默认情况下，ZLib 与您的应用程序静态链接，因此无需部署 ZLib 库。

如果您想使用特定库，请在项目中添加以下条件定义：

```
SGC_DYNAMICLOAD_ZLIB
```

作为替代方案，您可以编辑位于源文件夹中的 sgclndy.inc 文件，并添加以下行

```
{$DEFINE SGC_DYNAMICLOAD_ZLIB}
```

最后，您必须设置 ZLib 库的位置，为此，请使用以下方法并传入库所在位置的完整路径（不包含库名称）

```
sgclZLibHeaders.ldZLibSetLibPath('c:\software\zlib');
```

**此配置仅适用于带源代码的 sgcWebSockets 企业版。*

快速入门

WebSockets 组件

创建新的 WebSocket 服务器或 WebSocket 客户端非常简单，只需创建类的新实例，配置 Host/Port，然后将属性 Active = true 设置为启动进程即可。

[QuickStart WebSockets](#)

HTTP 组件

HTTP/2 协议允许您创建比 HTTP/1 协议快得多的 HTTP 服务器/客户端。HTTP/2 服务器包含在 WebSocket 服务器中，而 HTTP/2 客户端是实现 HTTP/2 协议的专用组件。

[快速入门 HTTP](#)

线程流程

sgcWebSockets 组件是多线程的，这意味着**连接在辅助线程中运行**。默认情况下，主要事件在主线程上分发，当需要分发的事件数量较少时这非常有用，但为了**更好的性能**，您可以配置组件，使**事件在连接线程上下文中分发**。请阅读以下文章，了解如何配置线程流程：

[如何配置 NotifyEvents](#)

如何构建应用程序

使用 sgcWebSockets 库构建应用程序非常简单。只需遵循 以下提示，它们将帮助您**成功 构建您的应用程序**。

[构建](#)

高性能服务器

sgcWebSockets 提供 **2 种服务器实现**：一种基于 **Indy 服务器**，另一种基于 **HTTP.SYS Microsoft 服务器**。后者推荐用于需要处理数千个并发连接的高性能服务器。请参阅以下文章，了解如何提升服务器性能。

[高性能服务器](#)

内存管理器

选择合适的内存管理器 可以提升您的应用程序性能。请参阅以下文章， 其中对比了多种内存管理器的性能。

[内存管理器](#)

OpenSSL

当您的应用程序需要安全连接时，通常需要 **openssl 库**来**加密 通信**，请按照以下步骤成功配置您的应用程序与 openssl 库。

[配置 OpenSSL](#)

Indy

Indy 库被用作某些 sgcWebSockets 组件的基础，sgcWebSockets Enterprise 版包含一个自定义 Indy 版本，允许您使用 openssl 1.1.1 和 openssl 3.0.0、ALPN 等。

[Indy](#)

Linux (Lazarus)

若您在 Linux 下编译 Lazarus 项目时收到以下消息：

信号量初始化失败（可能并发线程数过多）

只需将 **cthreads** 单元添加到您的项目文件中即可。

快速入门 | WebSockets

让我们从一个基本示例开始，我们需要创建一个服务器 WebSocket 和两种类型的客户端 WebSocket：应用程序客户端和 Web 浏览器客户端。

WebSocket 服务器

1. 创建一个新的 Windows 窗体应用程序
2. 将 TsgcWebSocketServer 拖放到窗体上。
3. 在事件选项卡中，双击 OnMessage 事件，并输入以下代码：

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

4. 将一个按钮拖放到窗体上，双击并输入以下代码：

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

WebSocket 客户端

1. 创建一个新的 Windows 窗体应用程序
2. 将 TsgcWebSocketClient 拖放到窗体上，并配置 Host 和 Port 属性以连接到服务器。
3. 在表单中拖放一个 TButton，双击并输入以下代码：

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

4. 将一个按钮拖放到窗体上，双击并输入以下代码：

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

Web 浏览器客户端

1. 创建一个新的 HTML 文件
2. 用文本编辑器打开文件并复制以下代码：

```
<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>
```

您需要在此文件中将 host 和 port 替换为您自定义的主机名和端口！

3. 保存文件，配置完成，您已成功配置了一个基本的 WebSocket Web 浏览器客户端。

使用方法

1. 启动服务器应用程序并按下按钮，以启动 WebSocket 服务器监听新连接。
2. 启动客户端应用程序，按 button1 连接到服务器，再按 button2 发送消息。在服务器端，您将看到客户端发送的文本消息。
3. 用网页浏览器（Chrome、Firefox、Safari 或 Internet Explorer 10+）打开 HTML 文件，点击"打开"建立连接，然后点击"发送"向服务器发送消息。在服务器端，您将看到一条由 Web 浏览器客户端发送的文本消息。

Linux 编译器

简单服务器示例（在端口 5000 上监听）。

```
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;
  end.
```

Linux (Lazarus)

若您在 Linux 下编译 Lazarus 项目时收到以下消息：

信号量初始化失败（可能并发线程数过多）

只需将 **cthreads** 单元添加到您的项目文件中。

快速入门 | HTTP

我们从一个基本示例开始，该示例需要创建一个 HTTP/2 服务器和一个 HTTP/2 客户端。

HTTP/2 服务器

1. 创建一个新的 Windows 窗体应用程序
2. 将 TsgcWebSocketHTTPServer 拖放到窗体上。
3. 在事件选项卡上，双击 OnCommandGet 事件，然后输入以下代码：

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

4. 默认情况下，服务器仅启用 HTTP/1 连接，因此请在属性 **HTTP2Options.Enabled = true** 中启用 HTTP/2 选项，然后配置 SSL 选项。安全连接需要 [OpenSSL 库](#)。

```
TsgcWebSocketHTTPServer1->Port = 443;
TsgcWebSocketHTTPServer1->SSL = true;
TsgcWebSocketHTTPServer1->SSLOptions->CertFile = "server cert file";
TsgcWebSocketHTTPServer1->SSLOptions->KeyFile = "server private key file";
TsgcWebSocketHTTPServer1->SSLOptions->RootCertFile = "server root cert file";
TsgcWebSocketHTTPServer1->SSLOptions->OpenSSL_Options->APIVersion = sslAPI_1_1;
TsgcWebSocketHTTPServer1->SSLOptions->Port = 443;
TsgcWebSocketHTTPServer1->SSLOptions->Version = tls1_3;
```

5. 在表单上放置一个 Button，双击并输入以下代码：

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

HTTP/1 客户端

1. 创建一个新的 Windows 窗体应用程序
2. 在窗体上拖放一个 TButton，双击并输入以下代码：

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

HTTP/2 客户端

1. 创建一个新的 Windows 窗体应用程序
2. 在窗体上拖放一个 TButton，双击并输入以下代码：

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

QuickStart | 线程流程

sgcWebSockets 组件是多线程的，例如，**TsgcWebSocketHTTPServer**（基于 Indy 库）为每个连接创建一个线程，而 **TsgcWebSocketServer_HTTPAPI**（基于 Microsoft HTTP.SYS）运行线程池，连接由该线程池处理（最多 64 个线程），**TsgcWebSocketClient** 运行自己的线程，以异步方式处理来自 WebSocket 服务器的响应。

默认情况下，有一个名为 **NotifyEvents** 的属性，其值为 **neAsynchronous**。这意味着当 WebSocket 客户端收到消息时，该消息会被排队，并由操作系统稍后在主线程上分派。这对于消息量不大的客户端非常适用，且易于使用，因为在需要更新表单上的控件时，无需进行线程同步。

但是，当服务器/客户端必须在短时间内处理多条消息时，最好将此线程流程更改为在连接线程上下文中分发事件的模式。为此，只需将 **NotifyEvents** 属性设置为 **neNoSync**，这样，例如当客户端从服务器收到消息时，该消息将在辅助线程的上下文中分发，因此如果需要更新窗体上的控件，请先与主线程同步，然后再更新窗体控件（因为窗体控件不是线程安全的）。同样，如果需要访问共享对象，也需要实现自己的同步方法。

线程流简单模式（NotifyEvents = neAsynchronous）与低性能

这是默认的线程流程，通常用于演示示例。如果您不需要每秒处理多条消息，且需要更新表单控件或访问共享对象，请选择此模式。

NotifyEvents = neAsynchronous

线程流最佳性能（NotifyEvents = neNoSync）

为服务器组件和需要高性能（因预期需要处理多条消息）的客户端设置此线程流程。使用此配置时，事件在连接线程的上下文中分发，因此若要更新窗体控件，请先与主线程同步。

NotifyEvents = neNoSync

如何同步 主线程

您可以通过调用 **TThread.Synchronize** 或 **TThread.Queue** 与主线程同步，两种方法均可使用，选择哪种取决于您希望如何实现同步。

TThread.Synchronize

此方法为阻塞式，即调用 **Synchronize** 时，代码将阻塞直到与主线程同步完成。

TThread.Queue

此方法为非阻塞方法，因此调用队列时，消息将被加入队列并在稍后分发。

示例代码

使用从 WebSocket 客户端 收到的消息更新 Memo。

```
void OnClientMessage(TsgcWSConnection *Connection, string aText)
{
    TThread::Queue(NULL,
        [&]() {
            memo1->lines->add(aText);
        });
}
```

QuickStart | 构建

使用 sgcWebSockets 库构建应用程序非常简单。只需注意您的组件是否需要 OpenSSL 库即可。如果您的应用程序需要安全连接，则必须部署 OpenSSL 库（在客户端组件上使用 [SChannel for Windows](#) 时除外）。

对于 **Windows 应用程序**，只需将 OpenSSL 库部署到应用程序所在的同一文件夹中即可。

有关其他特性，请参阅以下文章：

- [构建 OSX 应用程序](#)
- [构建 Android 应用程序](#)
- [构建 iOS 应用程序](#)

CBuilder DEBUG

如果您使用 CBuilder 并希望调试 sgcWebSockets 库，请按照以下步骤操作：

- 进入项目选项。
- 在 Delphi 编译器 > 编译中启用"使用调试 .dcu"。
- 在 C++ 链接器中禁用动态 RTL 链接。
- 在"包">"运行时包"下，禁用"与运行时包链接"。

然后您就可以逐步进入 VCL/RTL 源代码。

构建 | OSX 应用程序

要使用 sgcWebSockets 库构建 OSX 应用程序，您必须按照 Embarcadero 网站上的步骤构建 OSX 应用程序。

在 MacOS 上安装 PASServer

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

获取开发者 ID 证书

使用有效的 Apple Developer 账户登录 <https://developer.apple.com>，并从 Certificates 菜单创建新的"Developer ID Application"。

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

创建新的 Apple ID

然后访问 <https://appleid.apple.com/account> 创建新的 Apple ID

配置预配

最后，打开菜单 **Project / Options / Provisioning** 并填写所需数据以对 OSX 应用程序进行公证。

Provisioning

Target
Release configuration - macOS 64-bit platform

Build type
macOS 64-bit - Developer ID

Apple ID
your@email.com

App-specific Password
.....

Developer ID Application Certificate
Developer ID Application: *****

Additional options to pass to the notarization command-line tool

☐ Attach a ticket to the notarized application to allow it to run offline

如果您的项目需要某些库，请不要忘记在菜单 **Project / Deployment** 中包含它们。将远程路径设置为 "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_PROJECTIONCNS.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

当应用程序经过公证时，这些库将自动签名，您可以使用以下命令检查库是否已签名：

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

阅读更多关于如何[配置 OpenSSL OSX](#)。

构建 | Android 应用程序

要使用 sgcWebSockets 库构建 Android 应用程序，您必须遵循 Embarcadero 网站上构建 Android 应用程序的步骤。

创建 Android 应用

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

项目部署

如果您的项目需要某些库，请不要忘记在菜单 **Project / Deployment** 中包含它们。将远程路径设置为".\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

了解更多关于如何配置 [openSSL Android](#) 的信息。

构建 | iOS 应用程序

要使用 sgcWebSockets 库构建 iOS 应用程序，您必须按照 Embarcadero 网站上的步骤来构建 iOS 应用程序。

在 MacOS 上安装 PASServer

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

获取 iOS 开发证书

使用有效的 Apple 开发者账号登录 <https://developer.apple.com>，并从证书菜单创建新的"iOS 开发证书"。为您的 iOS 应用创建一个新的标识符，并创建一个新的配置文件。

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

配置 Bundle Identifier

打开菜单 **Project / Options / Application / Version Info** 并设置您的 Bundle Identifier

Version Info

Target
Release configuration - iOS Device 64-bit platform

☒ Include version information in project

Module version number

Major version	Minor version	Build
1	0	0

Build number options
Do not change build number

Key	Value
CFBundleName	\$(ModuleName)
CFBundleDevelopmentRegion	en
CFBundleDisplayName	\$(ModuleName)
CFBundleIdentifier	com.esegce.\$(ModuleName)
CFBundleInfoDictionaryVersion	6.0
CFBundleVersion	1.0.0
CFBundlePackageType	APPL
CFBundleSignature	????
LSRequiresiPhoneOS	true

部署

如果您的项目需要某些静态库，请将这些库复制到 Embarcadero lib/iosDevice64 文件夹：

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

了解更多关于如何 [配置 openssl iOS](#)

配置

最后，在菜单 **Project / Options / Deployment** 中检查证书是否已成功加载。

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

高性能服务器

基于 Indy 库的服务器

`TsgcWebSocketServer` 和 `TsgcWebSocketHTTPServer` 基于 Indy 库，每个连接由一个线程处理，因此 1000 个并发连接至少需要 1000 个线程。当性能至关重要时，您需要进行一些“调优”以提升性能并改善服务器运行状况。从 `sgcWebSockets 4.3.3` 起，Indy 服务器也支持 IOCP，您可以[阅读更多内容](#)。

使用以下技巧提高服务器性能。

1. 在服务器组件属性中设置 **NotifyEvents := neNoSync**。这意味着事件在连接线程的上下文中引发，因此没有同步机制。如果必须访问 VCL 控件或共享对象，请使用您自己的同步机制。

2. 在 Server 组件属性中设置 **Optimizations.Connections.Enabled := True**。如果您的服务器计划支持超过 1000 个并发连接，且频繁调用 `Server.WriteData` 方法，请启用此属性。该属性会将连接保存到缓存列表中，搜索速度比访问 Indy 连接列表更快。

2.1 CacheSize：存储在快速缓存中的连接数量。默认值 = 100。

2.2 GroupLevel：在内部创建多个按首字符分组的列表，因此在连接数较多时，搜索速度更快。默认值 = 1。

3. 在服务器组件属性中设置 **Optimizations.Channels.Enabled := True**。启用此属性后，频道将保存在列表中，搜索速度比之前的方法更快。

4. 在 Server 组件属性中设置 **Optimizations.ConnectionsFree.Enabled := True**。启用此属性后，每次断开连接时，`TsgcWSCConnection` 对象不会被销毁，而是存储在列表中，每隔 X 秒统一销毁。启用此属性会增加内存消耗，因此如果在短时间内有大量断开连接，请将此属性设为 false。

4.1 Interval：存储在列表中的所有已断开连接被销毁的时间间隔（秒）。默认值为 60。

5. 默认情况下，`sgcWebSockets` 使用**临界区**保护对共享对象的访问。但您也可以使用 `TMonitor` 或 `SpinLocks` 代替临界区。只需使用以下编译器定义之一编译项目即可。

3.1 {\$DEFINE SGC_SPINLOCKS}

3.2 {\$DEFINE SGC_TMONITOR}

6. 使用最新的 **FastMM4**，可从以下地址下载：<https://github.com/pleriche/FastMM4>

FastMM4 是一个非常好的内存管理器，但有时在多线程应用程序中扩展性不佳。请在您的应用程序中使用以下编译器定义：

{ \$DEFINE UseReleaseStack }

然后，将 FastMM4 添加为项目 `uses` 中的第一个单元，并重新编译。对于高并发服务器，您将注意到性能的提升。此调整的作用如下：如果在 `FreeMem` 调用期间无法立即释放一个块，该块将被添加到稍后释放的块列表中，在后台清理线程中或在下次调用 `FreeMem` 时释放。

7. 优于 FastMM4，请使用最新版 **FastMM5**，可从以下地址下载：<https://github.com/pleriche/FastMM5>

这是来自 FastMM4 同一开发者的新版本，支持 Delphi XE3 及更高版本，可在 Windows32 和 Windows64 上使用。

FastMM5 采用双重授权，包含 GPL 和商业两种许可证。若要在商业项目中使用，必须购买许可证。

以下表格对比了使用 FastMM4 和 FastMM5 时，以 1、10、100、500 和 1000 个并发客户端执行 10 万次 WebSocket 请求和响应的性能。在多线程应用程序中，FastMM5 的性能远优于 FastMM4。

客户端	Windows	FMM4	FMM5	差值
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 服务器 Windows

sgcWebSockets Enterprise Edition 在 Windows 上支持 **IOCP**，这意味着不再为每个连接创建 1 个线程，而是由线程池处理所有连接。要启用 IOCP，只需将 IOHandler 设置为 IOCP 即可。

```
IOHandlerOptions.IOHandlerType = iohIOCP
```

属性 IOHandlerOptions.IOCP 允许您自定义 IOCP 属性。

- **IOCPThreads**：用于处理连接的线程。默认值为零，表示线程将根据处理器数量自动计算（对于 Delphi 7 至 Delphi 2007，此值设置为 32）。
- **WorkOpThreads**：若希望每个连接的请求始终由同一线程处理，请设置大于零的值。默认情况下，IOCP 请求由随机线程处理；若希望连接始终由同一线程处理，请设置大于零的值。示例：若设置 WorkOpThreads = 32，服务器将创建 32 个线程，每次有新请求时，如果该连接之前已处理过，则将其排入同一线程的队列。

当您希望处理数千个并发连接时，推荐使用 IOCP。

Indy 服务器 Linux

sgcWebSockets Enterprise Edition 在 Linux 上支持 **EPOLL**，这意味着使用线程池处理所有连接，而不是为每个连接创建 1 个线程。要启用 EPOLL，只需将 IOHandler 设置为 EPOLL。

```
IOHandlerOptions.IOHandlerType = iohEPOLL
```

属性 IOHandlerOptions.EPOLL 允许自定义 EPOLL 属性。

- **EPOLLThreads**：这些是用于处理连接的线程，默认值为零，表示线程将使用处理器数量自动计算。
- **WorkOpThreads**：若希望每个连接的请求始终由同一线程处理，请设置大于零的值。默认情况下，EPOLL 请求由随机线程处理，若希望连接始终由同一线程处理，请设置大于零的值。示例：若 WorkOpThreads = 32，服务器将创建 32 个线程，每次有新请求时，若该连接之前已被处理，则将请求排队到同一线程。

当需要处理数千个并发连接时，推荐使用 EPOLL。

基于 HTTP.SYS 的服务器

[TsgcWebSocketServer_HTTPAPI](#) 组件基于 Microsoft HTTP API，设计为与 IOCP 协同工作，因此在服务器需要处理数千个连接时推荐使用，但它的限制是只能在 Windows 上运行。

服务器可以在同一端口上处理 **WebSocket** 和 **HTTP/2** 协议，并且可以与其他实现一起工作，因为它可以配置为只处理某些端点。

示例：您可以配置此服务器使用我们的 `sgcWebSockets` 库处理 WebSocket 连接，并让其他实现/第三方或任何人使用其他端点。

- 端点：`https://server/ws` 将使用 `sgcWebSockets` 处理通过 WebSocket 协议的连接
- 端点：`https://server/other` 将使用另一个库处理连接。

使用最新版 **FastMM5**，可从以下地址下载：<https://github.com/pleriche/FastMM5>

这是来自 FastMM4 同一开发者的新版本，支持 Delphi XE3 及更高版本，可在 Windows32 和 Windows64 上使用。

FastMM5 采用双重授权，包含 GPL 和商业两种许可证。若要在商业项目中使用，必须购买许可证。

以下网格对比了使用 FastMM4 和 FastMM5 时，以 1、10、100、500 和 1000 个并发客户端执行 10 万次 WebSocket 请求和响应的性能。在多线程应用程序中，FastMM5 的性能远优于 FastMM4。

客户端	Windows	FMM4	FMM5	差值
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%

内存管理器

最近，Pierre le Riche 开发的 FastMM 新版本已发布，新版本名为 **FastMM5**，经过重写以提升多线程应用程序的性能，可配置为更高速度或更低内存占用等。

支持 Delphi **XE3 编译器**及以上版本，可在 **Windows32** 和 **Windows64** 上使用。

FastMM5 采用**双重许可**，共有两种许可证：**GPL** 和**商业许可证**。若要在商业项目中使用，必须购买许可证。详情请见此处。

<https://github.com/pleriche/FastMM5>

FastMM4 有一个新的分支，名为 **FastMM4-AVX**，由 Maxim Masiutin 开发，增加了许多有趣的功能，例如：更高效的同步、使用 AVX 指令加速内存复制、速度提升等。FastMM4-AVX 采用双重许可：MPL 和 GPL。更多详情请参见：

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

配置

为了使用我们的组件测试性能，我们创建了一个新的 Windows 控制台应用程序 **sgcBenchmark.exe**，用于衡量每种内存管理器与我们 sgcWebSockets 组件配合使用时的性能。

测试非常简单，一个客户端（或多个客户端）连接到服务器，向服务器发送消息，服务器将相同消息回复给客户端。重复此过程 100,000 次。测试重复多次，改变并发客户端数量，先 1，然后 10、100.....测量的时间是客户端发送第一条消息到从服务器收到最后一条消息之间经过的时间（因此不包括连接服务器所用的时间）。

该基准测试将对比使用 Delphi 10.4.1 自带的默认内存管理器、FastMM5 和 FastMM4-AVX 的性能。

基准测试 Indy WebSocket 服务器

在第一个基准测试中，使用的服务器是 [Indy WebSocket 服务器](#)，该服务器基于 Indy TCP Server，因此每个连接创建 1 个线程。

值以毫秒为单位，例如在 Windows32 平台上使用 1 个客户端进行的第一次测试，使用默认内存管理器耗时 4135 毫秒，使用 FastMM5 耗时 4214 毫秒，使用 FastMM4-AVX 耗时 4823 毫秒。计算的百分比以参考值为基准，在本例中以 Delphi 自带的默认内存管理器为基准，百分比越低性能越好。

基准测试进行了 3 次，显示的值是基准测试总和除以 3 的结果。

对于基准测试，所用服务器为：

- Windows 2016 Server Datacenter
- 16 个虚拟处理器
- 32 GB RAM
- 2.2 GHz

所用 Delphi 版本为 Delphi 10.4.1，FastMM5 和 FastMM4-AVX 均从 GitHub 服务器获取最新版本。

以下是基准测试结果。

客户端	平台	默认值 (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%

HTTP.SYS 服务器基准测试

在第二个基准测试中，使用的服务器是 [HTTP.SYS WebSocket 服务器](#)，该服务器基于 HTTP API Microsoft Framework，连接由线程池处理。

数值以毫秒为单位，例如，在 Windows32 平台上使用默认内存管理器进行的第一个 1 个客户端测试耗时 5364 毫秒，使用 FastMM5 耗时 5182 毫秒，使用 FastMM4-AVX 耗时 5838 毫秒。百分比是相对于参考值（此处为 Delphi 自带的默认内存管理器）计算的，百分比越低，性能越好。

基准测试进行了 3 次，显示的值是基准测试总和除以 3 的结果。

对于基准测试，所用服务器为：

- Windows 2016 Server Datacenter
- 16 个虚拟处理器
- 32 GB RAM
- 2.2 GHz

所用 Delphi 版本为 Delphi 10.4.1，FastMM5 和 FastMM4-AVX 均从 GitHub 服务器获取最新版本。

以下是基准测试结果。

客户端	平台	默认值 (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%

基准测试说明

以下是对 3 种不同内存管理器进行基准测试后获得的结果的一些说明：

- 在单线程应用程序中，FastMM4、FastMM5 和 FasMM4-AVX 之间的性能差异不大。
- **FastMM5 和 FastMM4-AVX 在多线程应用程序中效果更好。**
- FastMM5 和 FastMM4-AVX 之间的**差异**在这些基准测试中**很小**。
- **Windows 32** 基准测试的性能**优于 Windows 64** 测试。在 Windows 64 应用程序中使用 FastMM5 或 FastMM4-AVX 比在 Windows 32 中带来的性能提升更为显著。

最终选择哪种内存管理器取决于具体项目。我认为没有哪种内存管理器在所有条件下都是最佳的，因此在选择之前，请反复测试，以确定哪种在您的需求下表现最佳。

OpenSSL

OpenSSL 是一个软件库，供需要通过计算机网络进行安全通信以防止窃听，或需要识别对端身份的应用程序使用。它被互联网服务器广泛使用，包括绝大多数 HTTPS 网站。

此库是基于 Indy 库的组件在需要安全连接时所必需的。如果您的应用程序需要 OpenSSL，则必须在部署应用程序之前在文件系统中准备好必要的文件：

目前，sgcWebSockets 支持：**1.0.2**、**1.1** 以及 **3.0 至 3.3 openssl** 版本。

平台	API 1.0	API 1.1	API 3.*	静态/动态链接
Windows (32 位和 64 位)	libeay32.dll 和 ssleay32.dll	libcrypto-1_1.dll 和 libssl-1_1.dll	libcrypto-3.dll and libssl-3.dll	动态
OSX	libcrypto.dylib, libssl.dylib	libcrypto.1.1.dylib, libssl.1.1.dylib	libcrypto.3.dylib, libssl.3.dylib	动态
iOS 设备 (32 位和 64 位)	libcrypto.a and libssl.a	libcrypto.a and libssl.a	libcrypto.a and libssl.a	Static
iOS 模拟器	libcrypto.dylib, libssl.dylib	libcrypto.1.1.dylib, libssl.1.1.dylib	libcrypto.3.dylib, libssl.3.dylib	动态
Android 设备	libcrypto.so, libssl.so	libcrypto.so, libssl.so	libcrypto.so, libssl.so	动态

请参阅以下内容，了解如何为每个平台 **配置 OpenSSL** 库：

- [Windows](#)
- [OSX](#)
- [Android](#)
- [iOS](#)

openssl 配置

sgcWebSockets 基于 Indy 的组件允许您配置部分 OpenSSL 属性。可访问以下属性：

- **服务器组件**：SSLOptions.OpenSSL_Options。
- **客户端组件**：TLSOptions.OpenSSL_Options。

API 版本

标准 Indy 库仅允许加载 **1.0.2 OpenSSL** 库；这些库已被弃用，最新的 OpenSSL 版本使用 1.1.1 API。**sgcWebSockets Enterprise** 允许您加载 **1.1.1 openssl** 库，您可以在此属性中配置将加载哪个 openssl API 版本。每个进程只能加载一个 API 版本（因此同一应用程序中不能混用 openssl 1.0.2 和 1.1.1 库）。

LibPath

此属性允许您设置 openssl 库的位置。这对 Android 或 OSX 项目非常有用，因为在这些平台上必须设置 openssl 库的位置。

接受以下值：

- **oslpNone**：此值不设置任何库路径值（这是默认值）。
- **oslpDefaultFolder**：此值设置 OpenSSL 库的默认文件夹。不同的 IDE 版本（Windows、OSX 等）此路径各不相同。

加载附加的 OpenSSL 函数

使用回调加载默认情况下未定义的额外 OpenSSL 函数。您可以在 [OpenSSL 加载额外函数](#) 中了解更多信息。

密码套件

如果您想在服务器上提供 TLS 1.2 和 1.3 支持，并使用最佳安全性和性能，请使用以下配置：

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

并设置以下密码列表。

```
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
```

自签名证书

您可以使用自签名证书进行测试。只需执行以下命令即可创建自签名证书：

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

该命令将创建 2 个文件：cert.pem（证书）和 key.pem（私钥）。您可以将两个文件合并为一个文件。只需创建一个新文件，将两个文件的内容复制进去即可，结构如下：

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

常见错误

SSL_GET_RECORD: 版本号错误

此错误表示服务器与客户端使用了不同版本的 SSL/TLS 协议。要修复此问题，请尝试在服务器和/或客户端组件中设置正确的版本。

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

SSL3_GET_RECORD: decryption failed or bad record mac

通常此错误在以下情况下触发：

1. 检查您使用的 OpenSSL 版本是否为最新版。若版本过旧，请更新至最新支持版本。
2. 若此错误随机出现，通常是因为多个线程正在访问同一个 OpenSSL 连接。您可以尝试将 NotifyEvents 设置为 neNoSync，这意味着 OnConnect、OnDisconnect、OnMessage 等事件将在连接线程的上下文中触发，可避免某些

同步问题并提供更好的性能。但需注意，如果您在收到消息时更新窗体中的可视控件，必须自行实现同步方法，因为可视控件不是线程安全的。

OpenSSL | Windows

有 32 位和 64 位两个版本。您必须将这些库复制到应用程序所在的同一文件夹中，或放入系统路径。

如果您的操作系统是 Windows 32 位，只需复制到 System32 文件夹即可。

如果您的操作系统是 64 位 Windows，请将 64 位版本复制到 System32 文件夹，将 32 位版本复制到 SysWow64 文件夹。

API 1.0

需要以下库：

- libeay32.dll
- ssleay32.dll

如果您的操作系统是 Windows 32 位，只需复制到 System32 文件夹即可。

如果您的操作系统是 64 位 Windows，请将 64 位版本复制到 System32 文件夹，将 32 位版本复制到 SysWow64 文件夹。

您可以从账户下载最新的库（库不包含外部依赖项，且已进行数字签名）。

API 1.1

需要以下库：

Windows 32

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

Windows 64

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

如果您的操作系统是 Windows 32 位，只需复制到 System32 文件夹即可。

如果您的操作系统是 64 位 Windows，请将 64 位版本复制到 System32 文件夹，将 32 位版本复制到 SysWow64 文件夹。

您可以从账户下载最新的库（库不包含外部依赖项，且已进行数字签名）。

API 3.*

需要以下库：

Windows 32

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

Windows 64

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

如果您的操作系统是 Windows 32 位，只需复制到 System32 文件夹即可。

如果您的操作系统是 64 位 Windows，请将 64 位版本复制到 System32 文件夹，将 32 位版本复制到 SysWow64 文件夹。

您可以从账户下载最新的库（库不包含外部依赖项，且已进行数字签名）。

如果您使用 p12 证书，则需要部署 legacy.dll 库。请阅读关于 [OpenSSL p12 证书](#) 的详细信息。

OpenSSL | OSX

较新版本的 OSX 不包含 OpenSSL 库，或版本过旧，因此必须随应用程序一起部署。请按以下步骤部署这些库：

- 在您的项目中打开 Project/Deployment。
- 添加所需的库。
- 将 RemotePath 设置为 'Contents\Macos'。
- 将 openssl LibPath 配置为默认文件夹：
 - Client.TLSOptions.OpenSSL_Options.LibPath = oslpDefaultFolder.
 - Server.SSLOptions.OpenSSL_Options.LibPath = oslpDefaultFolder.

API 1.0

需要以下库：

- libcrypto.dylib
- libssl.dylib

您可以从您的账户下载最新的库文件。

API 1.1

需要以下库：

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

有一个 32 位版本和一个 64 位版本。您必须将这些库复制到应用程序所在的同一文件夹中。
您可以从您的账户下载最新的库文件。

API 3.0

需要以下库：

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

仅提供 64 位版本。您必须将这些库复制到应用程序所在的同一文件夹中。
您可以从您的账户下载最新的库文件。

如果在 OSX 应用程序中包含 OpenSSL 库，应用程序经过公证后，这些库将被签名，您可以使用以下命令进行验证：

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

请查看以下视频，了解如何使用 OpenSSL 库构建 MacOSX64 应用程序。

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

错误

客户端不应加载未版本化的 `libcrypto.dylib`，因为它没有稳定的 ABI。

在 MacOS Monterey+ 上，尝试加载 OpenSSL 库时可能会出现此错误，原因是首先尝试加载不带版本号的 OpenSSL 库（例如 `libcrypto.dylib`）。

要修复此错误，请在属性 **OpenSSL_Options.UnixSymLinks** 中设置值 **oslsSymLinksDontLoad**。这可以避免加载不带版本号的 OpenSSL 库。

OpenSSL | Android

较新版本的 Android 不包含 OpenSSL 库，或所包含的版本过旧，因此必须随应用程序一起部署。请按照以下步骤部署这些库：

- 在您的项目中打开 Project/Deployment。
- 添加所需的库。
- 设置 RemotePath = '.\assets\internal'。
- 将 openssl LibPath 配置为默认文件夹：
 - Client.TLSOptions.OpenSSL_Options.LibPath = oslpDefaultFolder.
 - Server.SSLOptions.OpenSSL_Options.LibPath = oslpDefaultFolder.

API 1.0

需要以下库：

- libcrypto.so
- libssl.so

您可以从您的账户下载最新的库文件。

在 **Android 64 位**上，使用 TLS 1.2 可能会引发以下错误：

INT_RSA_VERIFY:bad signature

这是一个 OpenSSL 错误，已在 API 1.1 中修复。

您可以尝试使用 TLS 1.0 或 TLS 1.1（如果服务器仍支持这些加密方法以避免此错误）。

API 1.1

需要以下库：

- libcrypto.so
- libssl.so

您可以从您的账户下载最新的库文件。

API 3.0

需要以下库：

- libcrypto.so
- libssl.so

您可以从您的账户下载最新的库文件。

OpenSSL | iOS

要在 64 位 iOS 设备上安装 OpenSSL，您必须将 libcrypto.a 和 libssl.a SSL 库文件复制到您的系统。下载 iOS OpenSSL .zip 文件，解压后在 \lib 目录中找到 .a 文件。您必须将 libcrypto.a 和 libssl.a SSL 库文件复制到以下目录：

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

将 **sgcIdSSLOpenSSLHeaders_static** (or **IdSSLOpenSSLHeaders_static** if your sgcWebSockets edition is not Enterprise) 单元添加到您的 uses 子句中。

若需要部署任何文件，可将 RemotePath = StartUp\Documents，并使用以下方法加载文件（需在 uses 子句中添加 System.IOUtils）：

TPath.GetDocumentsPath + PathDelim + <您的文件名>

不得通过 iOS 下的菜单 **Project/Deployment** 部署 openssl 库。

API 1.1

修改 **IdCompilerDefines.inc** 并在 IOS 部分启用 **SGC_OPENSSL_API_1_1**：

```
{ $IFDEF IOS }
{ $DEFINE HAS_getifaddrs }
{ $DEFINE USE_OPENSSL }
{ $IFDEF CPUARM }
// RLebeau: 对于 iOS 设备，OpenSSL 不能作为外部库使用，
// 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
// 将静态 OpenSSL 库添加到项目，然后引入
// IdSSLOpenSSLHeaders_static unit in their uses clause. It hooks up the
// IdSSLOpenSSLHeaders_单元使用的静态链接函数...
{ $DEFINE STATICLOAD_OPENSSL }
// sgc--> 为 openssl API 1.1 启用
{ $DEFINE SGC_OPENSSL_API_1_1 }
{ $ENDIF }
{ $ENDIF }
```

您可以从您的账户下载库。

API 3.0

修改 **IdCompilerDefines.inc**，在 IOS 部分启用 **SGC_OPENSSL_API_1_1** 和 **SGC_OPENSSL_API_3_0**：

```
{ $IFDEF IOS }
{ $DEFINE HAS_getifaddrs }
{ $DEFINE USE_OPENSSL }
{ $IFDEF CPUARM }
// RLebeau: 对于 iOS 设备，OpenSSL 不能作为外部库使用，
// 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
// 将静态 OpenSSL 库添加到项目，然后引入
// IdSSLOpenSSLHeaders_static unit in their uses clause. It hooks up the
// IdSSLOpenSSLHeaders_单元使用的静态链接函数...
{ $DEFINE STATICLOAD_OPENSSL }
// sgc--> 为 openssl API 1.1 启用
```

```
{ $DEFINE SGC_OPENSSL_API_1_1 }  
// sgc --> 为 openssl API 3.0 启用  
{ $DEFINE SGC_OPENSSL_API_3_0 }  
{ $ENDIF }  
{ $ENDIF }
```

您可以从您的账户下载库。

OpenSSL | 自有 CA 证书

[Github 帖子](#)

创建由您自己的 CA 签名、且在将 CA 证书添加到本地机器后可被 Web 浏览器（如 Chrome）信任的证书。

1. 准备配置文件以便无提示地创建证书

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. 生成 CA 私钥和证书（有效期 5 年）

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

3. 生成 Web 服务器密钥和 CSR

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

4. 创建证书并由自有证书颁发机构签名（有效期 1 年）

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

5. 输出文件将为：

- CA.cnf → OpenSSL CA 配置文件。证书创建完成后可删除。
- CA_cert.pem → [证书颁发机构] 证书。必须将此证书添加到浏览器的本地颁发机构存储，以信任使用该 CA 创建的所有证书。
- CA_cert.srl → 随机序列号。可在证书创建过程完成后删除。
- CA_key.pem → 创建新的 [localhost] 证书时必须使用。证书创建完成后可以删除（如果您不打算重复使用它和 CA_cert.pem）。
- localhost.cnf → OpenSSL SSL 证书配置文件。证书创建过程完成后可以删除。
- localhost.csr → 证书签名请求。证书创建完成后可删除。
- localhost_cert.pem → SSL 证书。必须在服务器的 **SSLOptions.CertFile** 属性中进行配置。
- localhost_key.pem → 私钥。必须安装在服务器的 **SSLOptions.KeyFile** 属性中。

OpenSSL | P12 证书

OpenSSL 3.0 将多个已弃用或不安全的算法移入了一个名为 legacy provider 的内部库模块。该模块默认不加载，因此使用 OpenSSL 执行加密操作的应用程序（或其语言运行时）在加载证书、创建消息摘要时无法使用这些算法.....

旧版提供商中的算法包括 MD2、MD4、MDC2、RMD160、CAST5、BF（Blowfish）、IDEA、SEED、RC2、RC4、RC5 和 DES（但不含 3DES）。

出于安全原因，强烈建议停止使用这些旧版算法。

如果您的应用程序使用了以旧版密码（如 RC2-40-CBC）加密的客户端证书文件，可以使用 openssl 工具对证书文件进行重新加密以"现代化"处理。

例如，如果您的本地计算机上有一个 client.p12（或 client.pfx）证书文件：

```
$ openssl pkcs12 -legacy -in client.p12 -nodes -out cert-decrypted.tmp  
(如有提示，请输入密码短语)
```

```
$ openssl pkcs12 -in cert-decrypted.tmp -export -out client-new.p12  
(如有提示，请输入密码短语)
```

```
$ rm cert-decrypted.tmp
```

导出的 client-new.p12 证书文件现在包含相同的密钥，但使用 AES-256-CBC 加密。

请参阅以下 sgcWebSockets 和 sgclndy 包的配置：

sgcWebSockets

- 将属性 **OpenSSL_Options.Legacy.Enabled** 设为 True。
- 设置旧版库的位置。
 - **OpenSSL_Options.Legacy.LibPath**：在此配置旧版库的位置
 - **oslpNone**：这是默认值，旧版库应位于二进制文件所在的同一文件夹或已知路径中。
 - **oslpDefaultFolder**：自动设置旧版库路径，该路径是所有 IDE 版本中库应所在的位置。
 - **oslpCustomFolder**：如果选择此选项，请在 LibPathCustom 属性中定义完整路径。
 - **OpenSSL_Options.Legacy.LibPathCustom**：当 LibPath = oslpCustomFolder 时，在此处定义旧版库所在的完整路径。

sgclndy

- 将属性 **SSLOptions.Legacy** 设为 True。
- 在启动服务器或客户端之前，请设置 legacy.dll 库所在的路径。使用函数 **ldOpenSSLSetOSSLPPath** 并将路径作为参数传入。

OpenSSL | 验证证书

使用 OpenSSL 并设置"验证证书"选项时，可能会出现以下错误：

SSL 连接出错：error:80000002:system library::No such file or directory。
若您处理了 OnVerifyPeer 事件，且 Error 参数的值为 20，该错误表示：

```
X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY
```

此错误的主要原因是远程服务器提供的一个或多个证书不在您的应用程序的证书存储中。要解决此问题，您可以使用 RootCertFile 属性并设置 CA 文件所在的路径。如果您没有 CA 文件，可以从 Mozilla 等处下载：

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

设置 RootCertFile 后，之前的错误应已消除。

OpenSSL | 加载附加函数

默认情况下，Indy 定义了加密通信所需的最常用 OpenSSL 函数，但有时您需要更多函数用于加密、签名等。您可以使用 `IdOpenSSLSetLoadFuncsCallback` 方法分配一个回调，用于动态加载其他 OpenSSL 函数。

IdOpenSSLSetLoadFuncsCallback

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

这是一个过程类型，用作回调，接受三个参数：

- **hIdSSL**：TIdLibHandle — 已加载的 SSL 库句柄。
- **hIdCrypto**：TIdLibHandle — 已加载的 Crypto 库的句柄。
- **FailedLoadList**：TStringList，加载失败的函数列表。

此回调的目的是允许用户在加载 OpenSSL 函数时执行自定义处理。

IdOpenSSLSetUnLoadFuncsCallback

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

它作为卸载 SSL 函数的回调。当 OpenSSL 库被卸载时，可用于执行清理操作。

如何加载自定义函数

以下是如何使用回调加载 `EVP_PKEY_CTX_set_rsa_padding` 函数的简单示例。

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

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

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

void __fastcall DoOpenSSLUnLoadFuncsCallback() {
    EVP_PKEY_CTX_set_rsa_padding = nullptr;
}

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

// Mock function to simulate IdOpenSSLSetUnLoadFuncsCallback
void IdOpenSSLSetUnLoadFuncsCallback(void (__fastcall *callback)()) {
    callback(); // Example call
}

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

```
}
```

Indy

Indy 库是一个开源的客户端/服务器通信库，支持 TCP/UDP/RAW 套接字以及 100 多种高级协议，包括 SMTP、POP3、IMAP、NNTP、HTTP、FTP 等。Indy 使用 Delphi 编写，也适用于 C++Builder 和 FreePascal。sgcWebSockets 将 Indy 用作某些组件的基础，不同的 sgcWebSockets 版本对 Indy 库的使用方式不同。

sgcWebSockets 支持 HTTP/2 等需要 ALPN 的协议，可使用 openssl 1.1.1 或 openssl 3.0.0 使用 TLS 1.3.....所有这些功能都不被标准 Indy 库支持，因此 sgcWebSockets Enterprise 版本包含支持这些功能的自定义 Indy 库。为了避免从 IDE 中卸载标准 Indy 库，所需的 Indy 文件通过添加前缀"sgc"重命名，例如："IdGlobal" 单元重命名为 "sgcIdGlobal"。这样，两个版本可以共存而不会产生问题。

以下是每个 sgcWebSockets 版本使用的 Indy 版本：

sgcWebSockets 版本	Indy 版本
STANDARD	Standard
PROFESSIONAL	Standard
ENTERPRISE	自定义

拥有"已注册"许可证（即 sgcWebSockets 包拆分之前的旧许可证）的客户将找到以下 sgcWebSockets 包版本：

sgcWebSockets 版本	Indy 版本
sgcWebSockets	Standard
sgcWebSockets min	Standard
sgcWebSockets min Indy*	自定义

*sgcWebSockets_min_indy 与 sgcWebSockets 企业版相同。

自定义 Indy 版本的使用在源文件夹中的 "sgcVer.inc" 文件中定义。有一个名为 "SGC_CUSTOM_INDY" 的编译器定义，用于启用或禁用此 Indy 版本的使用。如果您拥有 Enterprise Edition 并希望禁用自定义 Indy 的使用，只需删除以下编译器定义：

```
{ $DEFINE SGC_CUSTOM_INDY }
```

当然，如果您启用了 SGC_CUSTOM_INDY 但源文件夹中没有所需的自定义 Indy 版本单元，此编译器定义将不起作用。

sgcIndy 包

自定义 Indy 版本的使用不局限于 sgcWebSockets 组件。eSeGeCe ALL-ACCESS 版本提供了一个附加的完整 Indy 包，包含支持 OpenSSL 1.1.1 和 OpenSSL 3.0.0 的完整 Indy 库。您必须先卸载 IDE 中当前安装的 Indy 库，然后再安装此版本（此处的单元未重命名）。

当您想使用 OpenSSL 库时，只需在加载 OpenSSL 库之前将全局变量 OPENSSL_API_VERSION 设置为所需的 OpenSSL API 版本。此全局变量位于 IdSSLOpenSSLHeaders 单元中。

示例：使用 openssl 1.1.1 库

```
OPENSSL_API_VERSION := openssl_1_1;
```

如何使用单个 sgclndy 包

当在同一应用程序中同时使用 eSeGeCe ALL-ACCESS 和 sgclndy 包时，源代码可能会重复，因为 sgWebSockets Enterprise 版本使用了自定义 Indy 版本，其中 Indy 单元已被重命名。这意味着，例如，sgclGlobal.pas 和 IdGlobal.pas 等单元将在同一应用程序中编译（前者在使用 sgWebSockets Enterprise 包的任何组件时使用，后者在使用 sgclndy 包的任何组件（如 ftp、smtp 等）时使用）。

为避免此行为，sgWebSockets 包可以配置为使用已安装的 sgclndy 版本，同时仍然使用所有组件。请按照以下说明操作：

1. 打开位于 sgWebSockets 包的 Source 文件夹中的 sgVer.inc 文件。
2. 禁用编译器指令：SGC_CUSTOM_INDY。此选项告知编译器以 sgclnd*.pas 开头的文件存在，并且在编译 sgWebSockets Enterprise 包时必须使用这些文件。
3. 启用以下编译器指令：SGC_INDY_LIB。此选项告知编译器 sgclndy 包已安装，编译包时必须使用它。

```
{ $DEFINE SGC_INDY_LIB }
```

使用上述配置，sgWebSockets Enterprise 包将使用已安装的 sgclndy 包，所有使用该包的功能（如 http/2、IOCP、openssl 3.0 等）都将启用。

WebSocket 事件

WebSocket 连接具有以下事件：

OnConnect

建立新连接时引发的事件。

OnDisconnect

连接关闭时触发的事件。

OnError

当连接发生任何错误时触发的事件。

OnMessage

收到新文本消息时触发的事件。

OnBinary

收到新二进制消息时触发的事件。

默认情况下，sgcWebSockets 使用**异步**机制触发这些事件，当任何这些事件在内部被触发时，它会将消息排入队列，并在操作系统允许时由其分发。此行为可通过名为 **NotifyEvents** 的属性进行修改，默认选择 **neAsynchronous**；如果勾选 **neNoSync**，则事件将在不与主线程同步的情况下触发（如果您需要更新任何 VCL 控件或访问共享资源，则需要实现自己的同步方法）。

neNoSync 在以下情况下推荐使用：

1. 您需要在极短的时间内处理大量消息。
2. 您的项目是为命令行构建的（如果不设置 **neNoSync**，您将不会收到任何事件）。
3. 您的项目是一个库。

如果不是，则可以使用 **neAsynchronous** 的默认属性值。

WebSocket 参数连接

支持 平台

[TsgcWebSocketClient](#)
JavaScript

有时，在建立新 WebSocket 连接时，将参数从客户端传递到服务器非常有用。如果需要向服务器传递一些参数，可以使用以下属性：

选项 / 参数

默认情况下设置为 '/'，如果需要传递参数（如 id=1），可将此属性设置为 '/?id=1'

在服务器端，您可以使用以下参数处理客户端参数：

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

使用 Javascript，您可以通过连接 URL 传递参数，示例：

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

在 DLL 内部使用

若您需要在动态链接库（DLL）和 `sgcWebSockets`（或控制台应用程序）中工作，**NotifyEvents** 属性需要设置为 **neNoSync**。

Windows 服务

在 Windows 服务中运行 sgcWebSockets 库时，**NotifyEvents** 属性必须设置为 **neNoSync**。

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

WebBrowser 测试

TsgcWebSocketServer 实现了一个内置网页，您可以在其中使用您喜欢的网络浏览器测试 WebSocket 服务器连接。

要访问此测试页面，您需要输入以下 URL：

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

示例：如果您已将 WebSocket 服务器配置在 IP 127.0.0.1 并使用端口 80，则需要输入：

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

在此页面中，您可以测试以下 WebSocket 方法：

打开
关闭
状态
发送

要禁用 WebBrowser HTML 测试页面，只需在 `TsgcWebSocketServer.Options.HTMLFiles` 中设置 `= false`；

自定义子协议

客户端可以通过在握手中包含子协议名称来请求服务器使用特定子协议。如果指定了，服务器需要在其响应中包含所选子协议值之一，才能建立连接。

要创建自己的子协议，您必须分别继承 `TsgcWSProtocol_Client_Base` 和 `TsgcWSProtocol_Server_Base` 来创建自定义子协议。



身份验证

支持方

TsgcWebSocketServer

TsgcWebSocketHTTPServer

TsgcWebSocketClient

Java script (*仅支持 URL 身份验证)

WebSocket 规范未定义任何身份验证方法，Web 浏览器实现不允许在新建 WebSocket 连接时发送自定义标头。

要启用此功能，您需要访问以下属性：

Authentication/ Enabled

sgcWebSockets 实现了 3 种不同类型的 WebSocket 身份验证：

会话：客户端需要通过 HTTP GET 传递用户名和密码，若验证通过，服务器将响应一个会话 ID。客户端使用此会话 ID 打开 WebSocket 连接并将其作为参数传递。您可以使用普通 HTTP 请求来获取会话 id，并将用户名和密码作为参数传递

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

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

这将返回一个令牌，用于通过 WebSocket 连接连接到服务器：

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

URL：客户端打开 WebSocket 连接，并将用户名和密码作为参数传递。

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

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

Basic：实现基本访问认证，仅适用于 VCL WebSocket（服务器和客户端）以及 HTTP 请求（Web 浏览器客户端不实现此类认证）。当客户端尝试连接时，它将使用 AUTH BASIC 规范发送一个请求头。

您可以使用 **Authentication/AuthUsers** 属性定义已认证用户列表。每条条目需按以下格式定义：user=password。
示例：

```
admin=admin
user=1234
....
```

有一个名为 **OnAuthentication** 的事件，当用户不在 AuthUsers 列表中、客户端未发送授权请求时，您可以在此处处理身份验证.....您可以检查 User 和 Password 参数，若正确则将 Authenticated 变量设为 True。示例：

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


安全连接

支持方

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketClient](#)
Web 浏览器

SSL 支持基于 Indy 实现，因此需要部署 openssl 库才能使用此功能。
TsgcWebSocketClient 支持 Microsoft SChannel，因此在 WebSocket 客户端中选择 SChannel 选项后，无需为 Windows 32 位和 64 位部署 openssl 库。

服务器端

要启用此功能，您需要启用以下属性：

SSL/ 启用

还有其他需要定义的属性：

SSLOptions/ CertFile/ KeyFile/ RootCertFile：您需要一个 .PEM 格式的证书来加密 WebSocket 通信。

SSLOptions/ Password：此项为可选，仅在证书有密码时需要。

SSLOptions/ Port：SSL 连接使用的端口。

客户端

要启用此功能，您需要启用以下属性：

TLS / 启用

OpenSSL

默认情况下，基于 Indy 的客户端和服务端组件在连接到安全 WebSocket 服务器时使用 OpenSSL 库。
Indy 仅支持 1.0.2 OpenSSL API，因此不支持 API 1.1。若您使用我们的自定义 Indy 库编译 sgcWebSockets，则可使用 API 1.1 并选择 TLS 1.3 版本。只需在 `OpenSSL_Options` 属性中选择您希望使用的 OpenSSL API 即可：

- **osIAPI_1_0**：这是默认的 Indy API，您可以使用带有 openssl 1.0.2 库的标准 Indy 包。
- **osIAPI_1_1**：仅在使用我们的自定义 Indy 库（企业版）编译 sgcWebSockets 时选择。将使用 openssl 1.1.1 库。
- **osIAPI_3_0**：仅在使用我们的自定义 Indy 库（Enterprise Edition）编译 sgcWebSockets 时才选择此选项。将使用 openssl 3.0.0 库。
 - **ECDHE**：允许您为 TLS 1.2 启用 ECDHE（更安全的连接）。

事件

有两个事件可用于自定义 SSL 设置：

OnSSLGetHandler

此事件在创建 SSL 处理程序之前引发，您可以在这里创建自己的 SSL 处理程序（需要继承自 `TIdServerIOHandlerSSLBase` 或 `TIdIOHandlerSSLBase`）并设置所需的属性。

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

OnSSLAfterCreateHandler

如果未创建自定义 SSL 对象，则默认使用 OpenSSL 处理程序创建一个。您可以根据需要访问 SSL 处理程序属性并进行修改。

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

Microsoft SChannel

从 sgcWebSockets 4.2.6 起，您可以使用 SChannel 代替 openssl（仅适用于 Windows 7 及以上版本的 Windows）。这意味着无需部署 openssl 库。Windows 7 支持 TLS 1.0，但如果您需要在 Windows 7 上使用 TLS 1.2 等更现代的实现，必须在 Windows 注册表中启用 TLS 1.1 和 TLS 1.2。需要 Delphi 2010 Professional 版（Delphi 7、2007 和 2009 需要 Enterprise 版）。

HeartBeat

支持平台

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketServer_HTTPAPI](#)
[TsgcWebSocketClient](#)

在服务器组件上，每隔 x 秒自动向所有活跃 WebSocket 连接发送一次 ping。

在客户端组件上，每隔 x 秒自动向服务器发送一次 ping。

HeartBeat 具有以下属性：

- **Enabled**：如果为 true，则发送 ping
- **Interval**：发送心跳的时间间隔（秒）。例如：若值为 10，则每隔 10 秒发送一次心跳。
- **超时**：等待服务器响应的时间。示例：若值为 30，则等待 30 秒接收响应，超时后关闭连接。
- **HeartBeatType**：允许自定义 HeartBeat 的工作方式
 - **hbtAlways**：每隔 Interval 中定义的 x 秒发送一次 ping。
 - **hbtOnlyIfNoMsgRcvInterval**：仅在 Interval 属性中定义的最近 x 秒内未收到消息时，每隔 x 秒发送一次 ping。

自定义心跳

客户端和服务组件允许自定义 HeartBeat，发送自定义 ping 并检查连接是否仍然存活。**OnBeforeHeartBeat** 事件正是为此而设计的；它允许您发送自定义消息，以及/或者不发送标准 ping。

示例：每 30 秒发送一条文本消息作为 ping。

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


WatchDog

支持 平台

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketServer_HTTPAPI](#)
[TsgcWebSocketClient](#)

服务器

在服务器组件上，在意外关闭后自动重新启动服务器。若要每 60 秒检查服务器是否活跃，只需设置以下属性：

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

WatchDog.Monitor 允许您验证新客户端是否可以连接到服务器。这通过一个内部客户端来完成，该客户端尝试打开与服务器的 WebSocket 连接；如果失败，则重启服务器。要以 10 秒超时监控客户端是否可以连接到服务器，请设置以下属性：

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

Client

在客户端组件上，意外断开后自动重新连接到服务器。要每 10 秒断开后重新连接，只需设置以下属性：

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

日志

支持方

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketClient](#)

这是一个用于调试 WebSocket 连接的实用功能，若要启用，请访问以下属性：

LogFile/ Enabled

启用后，每次建立新连接都会记录到文本文件中。在服务器组件上，如果文件不存在将自动创建，但在服务器关闭之前无法访问它。如果您希望在服务器活跃时打开日志文件，则需要在启动服务器之前创建日志文件。

示例：

```
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 消息

WebSocket 帧可以被掩码处理，这意味着记录的消息可能无法被读取。
当属性 **LogFile.UnMaskFrames** = True（默认值为 true）时

- 由 **WebSocket 客户端**发送的消息以**未掩码**方式保存。
- **WebSocket 服务器**接收到的消息以**掩码**和**无掩码**两种形式保存（原因是套接字读取缓冲区时尚不知道消息的协议，因此两种形式均保存）。

HTTP

支持方

[TsgcWebSocketHTTPServer](#)

TsgcWebSocketHTTPServer 是一个允许您使用同一端口处理 WebSocket 和 HTTP 连接的组件。当您需要搭建一个仅开放 HTTP 端口（通常为 80 端口）的服务器时，该组件非常实用。此组件支持所有 [TsgcWebSocketServer](#) 功能，并允许您提供 HTML 页面服务。

您可以使用 **DocumentRoot** 属性**静态提供 HTML 页面**，例如：将 test.html 保存在目录 "C:\inetpub\wwwroot" 中，并将 **DocumentRoot** 设置为 "C:\inetpub\wwwroot"。如果客户端尝试访问 test.html，它将被自动提供，示例：

<http://localhost/test.html>

或者，您可以通过代码**动态提供 HTML 或其他资源**，为此有一个名为 **OnCommandGet** 的事件，每次客户端请求新的 HTML 页面、图像、JavaScript 文件... 时触发。基本上，您需要检查客户端请求的文档（使用 `ARequestInfo.Document`），并向客户端发送响应（使用 `AResponseInfo.ContentText` 发送响应内容，`AResponseInfo.ContentType` 是响应类型，`AResponseInfo.ResponseNo` 是响应代码，通常为 200）。示例：

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

广播与频道

支持的组件

[TsgcWebSocketServer](#)

[TsgcWebSocketHTTPServer](#)

[TsgcWebSocketServer_HTTPAPI](#)

Broadcast 方法默认将消息发送给所有已连接的客户端，但您可以使用 **channels** 参数进行过滤，使消息仅广播给已订阅某个频道的客户端。

示例：假设服务器连接了两类客户端，即桌面端和移动端，您可以创建 "desktop" 和 "mobile" 两个频道。

如果您能在服务器的 OnConnect 事件中判断某个客户端是否为移动端，可以执行如下操作：

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

首先将 Connection 强制转换为 TsgcWSConnectionServer 以访问订阅方法，若该连接符合过滤条件，则将其订阅到 desktop 频道。频道的订阅可以在任意事件中进行。例如，您可以要求客户端告知其是否为移动端，并由客户端向服务器发送包含自身信息的消息。此后，您即可仅向桌面端连接广播消息：

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

如果共有 100 个连接，其中 30 个为移动端，则消息仅会发送给另外 70 个桌面端连接。

绑定

支持方

[TsgcWebSocketServer](#)

[TsgcWebSocketHTTPServer](#)

通常，服务器有多个 IP 地址。如果您启用 WebSocket 服务器并将监听端口设置为 80，则服务器启动时会尝试在所有 IP 的 80 端口上监听。因此，如果您有 3 个 IP 地址，它将在每个 IP 上绑定 80 端口。

绑定允许定义服务器使用的确切 IP 和端口。例如，如果您需要监听 IP 127.0.0.1（内部地址）和 80.254.21.11（公共地址）的 80 端口，可以在服务器激活之前进行如下配置：

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

上传大文件

支持的组件

[TsgcWebSocketHTTPServer](#)

[TsgcWebSocketServer_HTTPAPI](#)

当 HTTP 客户端发送 **multipart/form-data** 数据流时，服务器默认将其保存在内存中。当文件较大时，服务器可能会抛出**内存不足**异常。为避免此类异常，服务器提供了 `HTTPUploadFiles` 属性，允许您配置 POST 数据流的处理方式：内存流或文件流。若选择文件流方式，接收到的数据流将直接存储到硬盘，从而避免内存问题。

要将服务器配置为以文件流方式保存 `multipart/form-data` 数据流，请按照以下步骤操作：

1. 将属性 `HTTPUploadFiles.StreamType = pstFileStream` 设置好。使用此配置后，服务器将把这些数据流存储到硬盘。
2. 您可以配置将文件存储为文件流的**最小字节数**。默认值为零，即所有数据流均以文件流方式存储。
3. 通过 `SaveDirectory` 属性指定数据流的存储文件夹；若未设置，则存储在应用程序所在的目录。
4. 当客户端发送 `multipart/form-data` 时，内容以边界（boundary）编码。若启用了 `RemoveBoundaries` 属性，则在完整数据流接收完毕后，将自动提取边界中的内容。

示例代码

首先创建一个新的服务器实例，并将数据流设置为以文件流方式保存。

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

然后创建一个包含以下配置的 HTML 文件。

```
<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>
```

最后，用浏览器打开该 HTML 文件并向服务器发送一个文件。服务器将创建一个扩展名为 `.sgc_ps` 的新文件流，并在数据流完整接收后自动从边界中提取文件。

事件

以下 2 个事件可用于自定义文件上传流程（需启用 `HTTPUploadFiles.RemoveBoundaries` 属性）。

OnHTTPUploadBeforeSaveFile

该事件在文件保存之前触发，允许您自定义所接收文件的名称。

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

OnHTTPUploadAfterSaveFile

该事件在文件保存之后触发，允许您获知已保存文件的名称。

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

OnHTTPUploadReadInput

该事件在解码器读取到非文件类型的输入值时触发（例如：表单中包含 `name`、`date` 等字段变量时）。

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

压缩

支持方

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketClient](#)
Chrome 等网页浏览器

此功能在需要发送大量数据时非常有效，通常适用于二进制消息，因为它使用部分浏览器（如 Chrome）支持的 "PerMessage_Deflate" 协议对 WebSocket 消息进行压缩。

要启用此功能，需要激活以下属性：

Extensions/ PerMessage_Deflate / Enabled

当客户端尝试连接到 WebSocket 服务器且此属性已启用时，它会在请求头中包含此属性。如果服务器已激活此功能，它将向客户端发送已激活该协议的响应，之后所有消息都将被压缩。如果服务器不支持此功能，则所有消息将不经压缩发送。

在 Web 浏览器上，您无需执行任何操作。如果支持此扩展，它将自动使用；否则，消息将在不压缩的情况下发送。

如果 WebSocket 消息较小，最好不要启用此属性，因为它会消耗 CPU 周期来压缩/解压消息。但如果您正在传输大量数据，则会注意到消息交换速度的提升。

Flash

支持 平台

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)

WebSockets 受到众多 Web 浏览器的原生支持（请查阅 <http://caniuse.com/websockets>），但某些旧版本不支持 WebSockets（如 Internet Explorer 6、7、8 或 9）。您可以为这些不支持 WebSockets 的浏览器启用 **Flash 降级方案**。

几乎所有其他或较旧的浏览器都支持通过安装 Adobe Flash Player 来使用 Flash。为支持 Flash 连接，您需要在服务器上**开放端口 843**，因为 Flash 出于安全原因使用此端口检查跨域访问。如果端口 843 不可达，将等待 3 秒后尝试连接到服务器默认端口。

Flash 仅在浏览器不原生支持 WebSocket 时应用。因此，如果在服务器端启用 Flash Fallback，而浏览器原生支持 WebSocket，则仍将使用 WebSocket 作为传输。

要启用 Flash 回退，需要访问服务器上的 **FallBack / Flash** 属性并**启用**它。还有另外 2 个属性：

1. Domain：如果您需要将 Flash 连接限制到单个/多个域（默认情况下允许所有域）。示例：这将允许访问域名 swf.example.com

swf.example.com

2. 端口：如果需要将 Flash 连接限制到单个/多个端口（默认情况下允许所有端口）。示例：这将允许访问端口 123、456、457 和 458

123,456-458

Flash 连接仅支持文本消息，不支持二进制消息。

自定义对象

支持方

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketServer_HTTPAPI](#)
[TsgcWebSocketClient](#)

每次建立新的 WebSocket 连接时，sgcWebSockets 都会创建一个 [TsgcWSConnection](#) 类，您可以在其中访问标识符、收发字节数、客户端 IP 等属性，还有一个 **Data** 属性，可用于在内存中存储对象，如数据库访问、会话对象等。

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

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

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

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

群组

支持方

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketServer_HTTPAPI](#)

sgcWebSockets 提供了一种强大的方法，用于向指定的已连接客户端子集广播消息。一个群组可以包含任意数量的客户端，一个客户端也可以同时加入任意数量的群组。您无需显式创建群组。实际上，群组会在您第一次在 `Groups.Add` 调用中指定其名称时自动创建。

当您使用 **Groups.Add** 方法将用户添加到群组时，该用户将在当前连接期间接收定向到该群组的消息。

添加和删除用户

要向群组添加或移除用户，请调用 `Add` 或 `Remove` 方法，并传入群组名称和 `TsgcWSConnection` 类。当连接结束时，无需手动从群组中移除用户。

以下示例展示了 `Groups.Add` 方法的用法。

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

向群组发送消息

您可以如以下示例所示向群组所有成员发送消息。

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

或者，您可以向所有以"Room"开头的群组发送消息（因此，如果存在 `Room1`、`Room2`、`Room3`.....，这些用户将收到消息）。

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

事件

有两个事件可用于处理每次向群组添加或从群组移除客户端时的群组和客户端操作：

`OnClientAdded`
`OnClientRemoved`

示例：当成员离开群组时，向群组发送一条消息。

```
TsgcWebSocketServer1->Groups->OnClientRemoved() = OnClientRemovedEvent();  
  
void OnClientRemovedEvent(TObject *Sender, const TsgcwSServerGroupItem *aGroup,  
    const TsgcwSConnection *aConnection)  
{  
    aGroup->BroadCast("Client " + aConnection->Guid + " has disconnected");  
}
```

IOCP

支持者

[TsgcWebsocketServer](#)
[TsgcWebsocketHTTPServer](#)

***需要自定义 Indy 版本。**

Windows 下的 IOCP 是一种 API，允许使用有限的线程池处理数千个连接，而不是像 Indy 默认那样每个连接使用一个线程。

要为 Indy 服务器启用 IOCP，请转到 **IOHandlerOptions** 属性并选择 **iohIOCP** 作为 IOHandler 类型。

```
Server->IOHandlerOptions->IOHandlerType = iohIOCP;  
Server->IOHandlerOptions->IOCP->IOCPThreads = 0;  
Server->IOHandlerOptions->IOCP->WorkOpThreads = 0;
```

IOCPThreads 是用于 IOCP 异步请求（重叠操作）的线程。默认值为零，表示线程数根据处理器数量计算（Delphi 7 和 2007 除外，因为这些版本不支持 `cpucount` 函数，线程数固定为 32）。

WorkOpThreads 仅应在希望连接始终由同一线程处理时启用。使用 IOCP 时，请求由线程池处理，同一连接的每个请求可能由不同线程处理。如果希望每个连接始终由同一线程处理，请在 **WorkOpThreads** 中设置用于处理这些请求的线程数量。这会影响服务器性能，仅当确实需要此功能时才建议设置大于零的值。

当需要处理数千个连接时，建议为 Windows 服务器启用 IOCP。如果您的服务器最多只需处理 100 个并发连接，则可以沿用默认的 Indy 线程模型。

OnDisconnect 事件未触发

IOCP 的工作方式与默认的 Indy IOHandler 不同。对于默认的 Indy IOHandler，每个连接都在一个线程中运行，这些线程一直在运行，检查连接是否处于活跃状态，因此如果断开连接，会在短时间内得到通知。

IOCP 工作方式不同，它使用线程池处理所有连接，而非像 Indy 默认的每连接一个线程。在 IOCP 中，检测连接是否存活的唯一方式是尝试向套接字写入数据。若发生任何错误，则表示连接已关闭。有 2 种检测断连的方式：

1. 如果使用 **TsgcWebsocketClient**，可以在 **Options** 属性中启用它，**CleanDisconnect := True**（默认禁用）。启用后，客户端断开连接之前会发送一条通知服务器断开的消息，服务器将收到此消息并触发 **OnDisconnect** 事件。
2. 您可以在**服务器端**启用心跳，例如每 60 秒向所有已连接的客户端发送一次 ping，若有客户端断开，则调用 **OnDisconnect**。

EPOLL

支持者

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)

***需要 sgcWebSockets Enterprise Edition。**

Linux 的 EPOLL 是一种 API，允许使用有限的线程池处理数千个连接，而非像 Indy 默认的每连接一个线程方式。要为 Indy 服务器启用 EPOLL，请转至 **IOHandlerOptions** 属性，然后选择 **iohIEPOLL** 作为 IOHandler 类型。

```
Server->IOHandlerOptions->IOHandlerType = iohEPOLL;  
Server->IOHandlerOptions->EPOLL->EPOLLThreads = 0;  
Server->IOHandlerOptions->EPOLL->WorkOpThreads = 0;
```

EPOLLThreads 是用于 EPOLL 异步请求（重叠操作）的线程数，默认值为零，表示线程数根据处理器数量计算（Delphi 7 和 2007 除外，在这些版本中线程数设置为 32，因为不支持 `cpucount` 函数）。您可以手动调整线程数。
WorkOpThreads 仅在您希望连接始终在同一线程中处理时才需要启用。使用 EPOLL 时，请求由线程池处理，同一连接的每个请求可以在不同线程中处理。如果您希望在同一线程中处理每个连接，请在 **WorkOpThreads** 中设置用于处理这些请求的线程数。这会影响服务器的性能，只有在需要此功能时才建议将值设置为大于零。

当您需要处理数千个连接时，建议为 Linux 服务器启用 EPOLL。如果您的服务器最多只处理 100 个并发连接，则可以继续使用默认的 Indy 线程模型。

OnDisconnect 事件未触发

EPOLL 与默认的 Indy IOHandler 工作方式不同。使用默认 Indy IOHandler 时，每个连接都在一个线程中运行，这些线程持续运行并检查连接是否活跃，因此如果发生断连，会在较短时间内收到通知。

EPOLL 的工作方式与默认的 Indy（1 个线程对应 1 个连接）不同，它使用线程池来处理所有连接。对于 EPOLL，检测连接是否仍然存活的唯一方法是尝试向套接字写入数据，如果出现任何错误，则表示连接已关闭。有两种方式可以检测断线：

1. 如果使用 **TsgcWebSocketClient**，可以在 **Options** 属性中启用它，**CleanDisconnect := True**（默认禁用）。启用后，客户端断开连接之前会发送一条通知服务器断开的消息，服务器将收到此消息并触发 **OnDisconnect** 事件。
2. 您可以在**服务器端启用心跳**，例如每 60 秒向所有已连接的客户端发送一次 ping，若有客户端断开，则调用 **OnDisconnect**。

Linux 连接限制

若要增加并发打开的连接数，请使用以下命令

```
ulimit -n 10000
```

上一条命令将打开文件描述符的最大数量设置为 10000

ALPN

支持者

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketClient](#)

***需要自定义 Indy 版本。**

应用层协议协商（ALPN）是传输层安全（TLS）的一个扩展，用于应用层协议协商。ALPN 允许应用层以避免额外往返的方式协商在安全连接上使用哪种协议，且独立于应用层协议。安全 HTTP/2 连接需要它，因为 HTTP/2 相比 HTTP/1.x 改善了网页压缩并降低了延迟。

Client

您可以在 `TLSOptions.ALPNProtocols` 中配置客户端支持的协议。当客户端连接到服务器时，这些协议将在初始 TLS 握手的"Client Hello"中发送，列出客户端支持的协议，服务器将选择使用哪种协议（如有）。

您可以通过访问 `TsgcWSConnectionClient` 的 `ALPNProtocol` 属性来获取服务器选择的协议。

Server

当有新的 TLS 连接时，会调用 `OnSSLALPNSelect` 事件，在此您可以访问客户端支持的协议列表，服务器可从中选择支持的协议。

如果不支持任何协议，`aProtocol` 可以留空。

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

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

转发 HTTP 请求

支持 平台

TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI
TsgcWSHTTPWebBrokerBridgeServer
TsgcWSHTTP2WebBrokerBridgeServer
TsgcWSServer_HTTPAPI_WebBrokerBridge

您可以配置服务器将某些 HTTP 请求转发到另一台服务器。当您有多台服务器且只有一台服务器在公共地址上监听时，这非常有用。

示例：您可以配置服务器，将所有对 /internal 的请求转发到另一台服务器，而其他所有请求则由 sgcWebSockets 服务器处理。

使用 **OnBeforeForwardHTTP** 事件检查所请求的 URL 是否需要转发，若需要则设置转发目标 URL。

示例：如果您想将所有对文档"/internal"的请求转发到服务器"localhost:8080"，请执行以下操作：

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

其他选项

当您想要转发 HTTP 请求时，可以使用以下附加选项：

1. 默认情况下，请求使用原始文档转发。**示例：**如果您将请求 http://localhost:8080/internal 转发到内部服务器 http://localhost:5555，则转发的 URL 将为 http://localhost:5555/internal。但您可以使用 Forward 对象的 **Document** 属性修改文档（默认情况下，它将使用与原始请求相同的文档）。

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

2. 如果您转发安全 HTTP 连接（HTTPSs），可在 Forward 对象的 **TLSSOptions** 属性中自定义 SSL/TLS 选项。**示例：**设置 TLS 版本

```
aForward.TLSSOptions.Version = tls1_2
```

3. 可使用以下属性自定义 HTTP 请求：

- **QueryParams**：文档之后的参数，例如 'id=1&user=2'。
- **Host**：指定请求发送目标服务器的主机和端口号。示例：www.esegece.com:443
- **Origin**：引起请求的来源（协议、主机名和端口）。示例：https://www.esegece.com/document。
- **LogFilename**：用于存储请求/响应的文件名。
- **NoCache**：若请求不得使用 Web 浏览器缓存，默认情况下已启用。
- **CustomHeaders**：要添加到请求中的自定义标头列表。示例：CustomHeaders.Add('X-ReverseProxy-Host: http://127.0.0.1:8888/test');

服务质量

支持方

TsgcWSPServer_sgc
TsgcWSPClient_sgc
TsgcWSPClient_MQTT
JavaScript

SGC 默认协议和 MQTT 实现了消息投递的 QoS（服务质量）。共有 3 种类型：

Level 0："至多一次"，消息按底层 TCP/IP 网络的最大努力投递。可能发生消息丢失或重复。该级别可用于例如环境传感器数据的场景，因为单次读数的丢失无关紧要，下一次读数很快会被发布。

级别 1："至少一次"，消息保证到达，但可能出现重复。

第 2 级："精确一次"，消息保证恰好到达一次。例如，在计费系统中可使用此级别，因为重复或丢失的消息可能导致错误的收费。

Level 0

消息根据底层 TCP/IP 网络的尽力传输原则投递。协议中未定义响应期望和重试语义。消息到达服务器时，要么仅一次，要么根本不到达。

下表显示 QoS 级别 0 的协议流程。

Client	消息和方向	服务器
QoS = 0	PUBLISH ----->	操作：向订阅者发布消息

级别 1

服务器通过 ACKNOWLEDGEMENT 消息确认收到消息。如果通信链路或发送设备发生已识别的故障，或在指定时间内未收到确认消息，则发送方将重新发送消息。消息将至少到达服务器一次。

QoS 级别 1 的消息在消息中有消息 ID。

下表显示了 QoS 级别 1 的协议流程。

Client	消息和方向	服务器
QoS = 1 Message ID = x 操作：存储消息	PUBLISH ----->	操作： • 存储消息 • 向订阅者发布消息 • 删除消息
动作：丢弃消息	ACKNOWLEDGEMENT <-----	

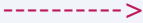
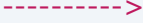
若客户端未收到确认消息（在应用程序定义的时间段内，或检测到故障且通信会话被重后），客户端可重新发送 PUBLISH 消息。

Level 2

QoS 级别 1 以上的附加协议流程确保不会向接收应用程序传递重复消息。这是最高级别的交付，适用于不能接受重复消息的情况。虽然网络流量会增加，但由于消息内容的重要性，这通常是可以接受的。

QoS 级别为 2 的消息在消息中包含消息 ID。

下表展示了 QoS 级别 2 的协议流程。对于接收方应如何处理 PUBLISH 流，有两种可用的语义。

Client	消息和方向	服务器
QoS = 2 消息 ID = x 操作：存储消息	PUBLISH 	Action： 存储消息
	PUBREC 	消息 ID = x
消息 ID = x	PUBREL 	操作： • 向订阅者发布消息 • 删除消息
动作：丢弃消息	确认应答 	消息 ID = x

如果检测到失败，或在定义的时间段之后，将从最后一条未确认的协议消息重新尝试协议流。额外的协议流确保消息只被投递给订阅者一次。

队列

支持方

TsgcWSPServer_sgc
TsgcWSPClient_sgc
JavaScript

SGC 默认协议 实现了队列以为已发布消息添加持久性（仅适用于**已发布消息**）

级别 0：消息不在服务器上排队

级别 1：服务器上仅队列最后一条消息，每次客户端订阅新频道或连接到服务器时发送。

级别 2：所有消息都在服务器上排队，并在每次客户端订阅新频道或连接到服务器时发送。

Level 0

消息未由服务器排队

下表显示队列级别 0 的协议流程。

Client	消息和方向	服务器
Queue = 0	PUBLISH ----->	操作： 向订阅者发布消息

级别 1

Level 1 的消息存储在服务器上，如果该频道已有其他存储的消息，则会被删除。

下表显示了 队列级别 1 协议流程。

Client	消息和方向	服务器
Queue = 1	PUBLISH ----->	操作： • 删除 此频道的所有消息 • 按频道 存储最后一条消息
动作： 处理消息	NOTIFY -----<	Action： 每次有新客户端订阅此频道时，都会发送最后一条消息。

这在发布者以"异常报告"方式发送消息时非常有用，即消息之间可能存在较长时间间隔。这允许新订阅者立即接收到保留的或最后已知的有效值数据。

Level 2

所有队列级别为 2 的消息都存储在服务器上。

下表展示了队列级别 2 的协议流程。

Client	消息和方向	服务器
Queue = 2	PUBLISH ----->	Action ：存储消息
动作 ：处理消息	NOTIFY <-----	Action ：每当有新客户端订阅此通道时，将发送所有消息。

事务

支持方

[TsgcWSPServer_sgc](#)
[TsgcWSPClient_sgc](#)
JavaScript

sgcWebSockets SGC 协议支持事务性 消息传递，当客户端提交事务时，所有客户端 发送的消息都在服务器端处理。客户端 调用 3 个方法：

StartTransaction

在服务器端创建新事务，此方法调用之后从客户端发送到服务器的所有消息都将在服务器端排队，直到客户端调用 Commit 或 Rollback。

Client	消息和方向	服务器
Channel = X	STARTTRANSACTION ----->	Action： 创建一个新队列以存储 指定频道的所有消息
Channel = X	PUBLISH ----->	操作： 消息存储在服务器端。
Action： 客户端收到消息 发送确认	ACKNOWLEDGEMENT < -----	Action： 由于消息已存储，服务器向客户端返回确认。
....		

提交

当客户端调用 Commit 时，服务器排队的所有消息都将被处理。

Client	消息和方向	服务器
Channel = X	COMMIT ----->	操作： 处理事务排队的所有消息

RollBack

当客户端调用 RollBack 时，服务器排队的所有消息将被删除，不在服务器端处理。

Client	消息和方向	服务器
Channel = X	ROLLBACK ----->	操作： 删除事务排队的所有消息

TCP 连接

支持者

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketClient](#)

默认情况下，sgcWebSocket 使用 WebSocket 作为协议，但您可以在客户端和服务端组件中使用普通 TCP 协议。

客户端组件

禁用 WebSocket 协议。

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

服务器组件

处理 OnUnknownProtocol 事件，将 Transport 设置为 trpTCP 并接受连接。

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

当客户端连接到服务器时，该连接将被定义为 TCP，并使用纯 TCP 协议而非 WebSocket。纯 TCP 连接不区分文本消息和二进制消息，因此所有接收到的消息都通过 OnBinary 事件处理。

消息结束

如果消息较大，有时可能以分片方式接收。可以通过指定要查找的字节来检测消息结尾。示例：在 STOMP 协议中，所有消息均以字节 0 和 10 结尾。

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

SubProtocol

支持者

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketClient](#)
[TsgcWebSocketServer_HTTPAPI](#)

WebSocket 提供简单的子协议协商。它基本上是在请求中添加一个包含支持的协议名称的头部。这些协议被接收端接收，如果接收端支持其中一个，则发送包含所支持子协议的响应。

sgcWebSockets 支持多种子协议：[MQTT](#)、[WAMP](#).....以及更多。您可以使用一种非常简单的方法实现自己的子协议，只需调用 RegisterProtocol 并将子协议名称作为参数传入即可。

示例：您需要连接到实现子协议 "Test 1.0" 的服务器

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

要在单个连接中使用多个协议，可以使用 **Broker Protocol**（服务器和客户端）组件来处理。只需在客户端/服务器与协议之间放置一个 Broker 即可。**示例：**通过单个连接同时使用 SGC 和 Files 协议。

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

当代理协议附加在服务器/客户端和协议之间时，**OnConnect** 和 **OnDisconnect** 事件在代理组件中触发（而不是在服务器或客户端组件中）。

Throttle

支持方

[TsgcWebSocketServer](#)

[TsgcWebSocketHTTPServer](#)

[TsgcWebSocketClient](#)

带宽限速功能受服务器和客户端组件支持，启用后可限制套接字每秒发送/接收的比特数。Indy 使用阻塞方法，因此如果客户端限制了读取速率，未读数据将停留在客户端套接字中，服务器将被阻塞无法向客户端写入新数据。客户端读取数据越慢，服务器写入新数据的速度也越慢。

服务器推送事件（推送通知）

支持平台

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
JavaScript

SSE 不是 WebSocket 的一部分，它定义了一个用于打开 HTTP 连接以接收来自服务器的推送通知的 API。

SSE 通过传统 HTTP 发送。这意味着它们不需要特殊协议或服务器实现即可使用。此外，Server-Sent Events 还具有 WebSockets 在设计上所缺乏的多种功能，例如自动重连、事件 ID 以及发送任意事件的能力。

事件

- **Open**：当新的 SSE 连接打开时。
- **Message**：当客户端收到新消息时。
- **错误**：当发生任何连接错误（如断开连接）时触发。

JavaScript API

要订阅事件流，请创建一个 EventSource 对象，并向其传入流的 URL：

```
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);
```

当从服务器推送更新时，onmessage 处理程序触发，新数据在其 e.data 属性中可用。如果连接关闭，浏览器将在约 3 秒后自动重新连接到源（这是默认重试间隔，您可以在服务器端更改）。

字段

以下字段名称由 规范定义：

event

事件的类型。若指定了此项，浏览器将向指定事件名称的监听器分发事件；网站需使用 addEventListener() 监听命名事件。若未为消息指定事件名称，则调用 onmessage 处理程序。

data

消息的 data 字段。当 EventSource 接收到多行连续以 data: 开头的内容时，将把它们拼接在一起，并在每行之间插入换行符。末尾的换行符会被移除。

id

要设置到 EventSource 对象最后事件 ID 值的事件 ID。

重试

尝试发送事件时使用的重连时间。必须是整数，指定重连时间（毫秒）。如果指定了非整数值，则忽略该字段。

所有其他字段名均被忽略。

对于多行字符串，请使用 #10 作为换行符。

使用示例：

若需向客户端发送消息，只需使用 WriteData 方法即可。

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

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

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

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

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

监听"notifications"频道的 JavaScript 代码：

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

LoadBalancing

支持方

[TsgcWebSocketServer](#)

[TsgcWebSocketHTTPServer](#)

[TsgcWebSocketLoadBalancerServer](#)

负载均衡允许在多台后端服务器之间分配工作。每次新客户端请求连接时，它连接到负载均衡器服务器（该服务器与后端服务器相连），并返回包含主机、端口等信息的连接字符串，客户端使用该字符串连接到服务器。例如，如果您有 4 台服务器，使用此方法所有服务器将拥有大致相同数量的连接，工作负载也将相近。

如果客户端想向所有服务器的所有客户端发送消息，只需使用 Broadcast 方法，消息将广播到连接到负载均衡服务器的所有服务器。

要启用此功能：

1. 放置一个 [TsgcWebSocketLoadBalancerServer](#) 组件，设置监听端口，并将 active 设为 True。
2. 服务器和客户端组件均有一个名为 LoadBalancer 的属性，您需要在其中设置负载均衡器服务器的主机和端口，并将 Enabled 设置为 True。

该组件允许对 **WebSocket** 和 **HTTP** 协议进行负载均衡。

文件

支持方

[TsgcWSPServer_sgc](#)

[TsgcWSPClient_sgc](#)

此协议允许以简便的方式在客户端与服务器之间传输文件。从小文件到大文件均可发送，且内存占用极低。您可以设置：

1. 数据包大小（字节）。
2. 使用自定义频道仅向已订阅的客户端发送文件。
3. 文件发送和接收的进度。
4. 对接收到的文件进行授权。
5. 已发送数据包的确认。

Proxy

支持方

[TsgcWebSocketClient](#)

客户端 WebSocket 组件支持通过 HTTP 代理建立 WebSocket 连接。要启用代理连接，需要激活以下属性：

代理 / 启用

设置为 True 后，您可以配置：

Host：代理服务器地址

Port：代理服务器端口

UserName/Password：连接代理的身份验证信息，仅在需要时填写。

ProxyType：支持以下代理类型：

- HTTP
- Socks4
- Socks4A
- Socks5

您可以通过访问 SOCKS 属性并将 Enabled 设置为 True 来配置 SOCKS 代理。

分片消息

支持方

[TsgcWebSocketServer](#)
[TsgcWebSocketHTTPServer](#)
[TsgcWebSocketClient](#)
[TsgcWebSocketServer_HTTPAPI](#)

默认情况下，当使用 sgcWebSockets 库发送流时，它会在单个数据包中发送所有数据，或者缓冲所有数据包，当收到最后一个数据包时，调用 OnBinary 消息事件。

此行为可通过 **Options.FragmentedMessages** 属性自定义，该属性接受以下值：

1. frgOnlyBuffer：这是默认值。表示数据包消息将被缓冲，只有当整个流接收完毕后，才会调用 OnBinary 消息事件。
2. frgOnlyFragmented：这意味着 OnFragmented 事件只会 对每个接收到的数据包调用一次。
3. frgAll：这意味着 OnFragmented 事件将在每个接收到的数据包时以及收到完整流时被调用。

OnFragmented 事件在您需要发送大型流且接收方必须显示传输进度时非常有用。

示例：客户端必须向服务器发送 1,000,000 字节的流，且服务器希望每收到 1000 字节就显示进度。

客户端将使用 writedata 方法发送一个数据流，包大小为 1000。

```
Client->WriteData(stream, 1000);
```

服务器将在 Options.FragmentedMessages 中设置 := frgAll，并处理 OnFragmented 事件以接收流的进度。

```
void OnFragmented(TsgcWSConnection *Connection, const TMemoryStream *Data, const TOpCode OpCode, const boolean C
{
    ShowProgress(Data->Size);
    if (Continuation == false)
    {
        SaveStream(Data);
    }
}
```

TsgcWebSocketClient

客户端 WebSocket 组件，可连接到任何 RFC 6455 WebSocket 服务器并收发文本和二进制消息。

简介

TsgcWebSocketClient 实现客户端 WebSocket 组件，可连接到 WebSocket 服务器。请按照以下步骤配置该组件：

1. 将 **TsgcWebSocketClient** 组件拖放到窗体上。
2. 设置 **Host** 和 **Port**（默认为 80）以连接到可用的 WebSocket 服务器。您可以设置 **URL** 属性，Host、Port、Parameters 等将从 URL 自动更新。**示例**：`wss://127.0.0.1:8080/ws/` 将产生如下结果：

```
oClient = new TsgcWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClient->TLS = true;
oClient->Options->Parameters = "/ws/";
```

3. 您可以选择是否需要 **TLS**（安全连接），默认情况下未激活。
4. 您可以通过 HTTP 代理服务器进行连接，需要定义代理属性：
Host：代理服务器主机名。
Port：代理服务器端口号。
Username：认证用的用户名，匿名访问留空。
密码：用于认证的密码，匿名登录时留空。
5. 如果服务器支持**压缩**，您可以启用压缩以对发送的消息进行压缩。
6. 设置允许的**规范**，默认情况下所有规范均已启用。
- RFC6455**：是标准且推荐的 WebSocket 规范。
- Hixie76**：始终为 false
7. 将 `Active = true` 设置为启动新的 WebSocket 连接。

参考

- 属性 — 组件上有 23 个可用属性。
- 方法 — 22 个公共方法。
- 事件 — 可处理 16 个事件。

指南

最常见的用途

- 连接
 - [如何连接到 WebSocket 服务器](#)
 - [打开客户端连接](#)
 - [关闭客户端连接](#)
 - [保持连接活跃](#)
 - [意外断连](#)
 - [连接 TCP 服务器](#)
 - [WebSocket 重定向](#)
- 安全服务器
 - [连接安全服务器](#)
 - [OpenSSL 证书](#)
 - [SChannel 证书](#)
 - [SChannel 获取连接信息](#)
- 发送消息

- 发送文本消息
- 发送二进制消息
- 接收消息
 - 接收文本消息
 - 接收二进制消息
- 身份验证
 - 客户端身份验证
- 其他
 - 客户端异常
 - 客户端 WebSocket 握手
 - 客户端注册协议
 - 客户端代理

TsgcWebSocketClient | 连接 WebSocket 服务器

URL 属性

连接到 WebSocket 服务器最简便的方式是使用 **URL** 属性并设置 **Active = true**。

示例：使用安全连接连接到 www.esegece.com。

```
oClient = new TsgcWebSocketClient();
oClient->URL = "wss://www.esegece.com:2053";
oClient->Active = true;
```

主机、端口和参数

您可以使用 **Host** 和 **port** 属性连接到 WebSocket 服务器。

示例：使用安全连接连接到 www.esegece.com

```
oClient = new TsgcWebSocketClient();
oClient->Host = "www.esegece.com";
oClient->Port = 2053;
oClient->TLS = true;
oClient->Active = true;
```

TsgcWebSocketClient | 客户端打开连接

配置客户端连接到服务器后，有 3 种不同的方式打开新连接。

Active 属性

打开新连接最简便的方法是将 Active 属性设为 true，这将使用组件配置尝试连接到服务器。如果将 Active 属性设为 false，则会在连接处于活动状态时关闭连接。此方法在与调用方相同的线程中执行。因此，如果您从主线程调用它，该方法将在应用程序的主线程中执行。

打开连接

```
oClient = new TsgcWebSocketClient();  
...  
oClient->Active = true;
```

当您调用 Active = true 时，**仍无法向服务器发送任何数据**，因为客户端可能仍在连接中。您必须首先等待 OnConnect 事件触发，然后才能开始向服务器发送消息。

关闭连接

```
oClient->Active = false;
```

当您调用 Active = false 时，**您无法确定连接在此代码执行后立即关闭**，因此必须等待 OnDisconnect 事件触发。

Start/Stop 方法

当您调用 Start() 或 Stop() 连接/断开服务器时，调用在辅助线程中执行，因此不会阻塞调用它的线程。如果您希望连接到服务器后让下面的代码继续执行，请使用此方法。

打开连接

```
oClient = new TsgcWebSocketClient();  
...  
oClient->Start();
```

调用 Start() 后，**仍然无法向服务器发送任何数据**，因为客户端可能仍在连接中。您必须先等待 OnConnect 事件触发，然后才能开始向服务器发送消息。

关闭连接

```
oClient->Stop();
```

调用 Stop() 后，**您无法确定连接是否已立即关闭**，因此必须等待 OnDisconnect 事件触发。

Connect/Disconnect 方法

当您调用 Connect() 或 Disconnect() 来打开/关闭与服务器的连接时，调用在被调用的同一线程中执行，但会等待直到过程完成。您必须设置一个 Timeout 以定义等待过程完成的最长时间（默认为 10 秒）。

示例：连接到服务器并等待最多 5 秒

```
oClient = new TsgcWebSocketClient();  
...  
if (oClient->Connect(5000) == true)  
{  
    oClient->WriteData("Hello from client");  
}  
else  
{  
    Error();  
}
```

如果 `Connect()` 方法返回成功结果，您已经可以向服务器发送消息，因为连接已建立。

示例：从服务器断开连接并等待最多 10 秒

```
if (oClient->Disconnect(10000) == true)  
{  
    ShowMessage("Disconnected");  
}  
else  
{  
    ShowMessage("Not Disconnected");  
}
```

如果 `Disconnect()` 方法返回成功结果，则表示连接已经关闭。

OnBeforeConnect 事件可用于在客户端尝试连接之前自定义服务器连接属性。

TsgcWebSocketClient | 客户端关闭连接

连接可通过 Active 属性、Stop 或 Disconnect 方法关闭，详情请参阅 [客户端打开连接](#)。

CleanDisconnect

当连接关闭时，您可以通过发送关闭消息来通知对端连接正在关闭。要启用此功能，请将 Options.CleanDisconnect 属性设置为 true。

如果启用此属性，在关闭连接之前将向服务器发送 Close 消息，通知服务器客户端正在关闭连接。

断开连接

[TsgcWSConnection](#) 有一个名为 Disconnect() 的方法，允许您在套接字级别断开连接。调用此方法后，套接字将直接断开连接，而无需等待服务器的任何响应。您可以在此方法中发送关闭代码。

关闭

[TsgcWSConnection](#) 有一个名为 Close() 的方法，允许您向服务器发送请求关闭连接的消息。如果服务器收到此消息，它必须关闭连接，客户端将收到连接已关闭的通知。您可以通过此方法发送关闭代码。

TsgcWebSocketClient | 保持客户端连接开启

客户端连接到服务器后，有时可能因信号差、连接错误等原因导致连接断开。有 2 个属性可帮助保持连接活跃。

HeartBeat

HeartBeat 属性允许您 每 X 秒 发送一次 Ping 以保持连接活跃。某些服务器在对等方之间无数据交换时会关闭 TCP 连接。HeartBeat 通过在特定间隔发送 ping 来解决此问题。通常这足以维持连接活跃，但如果希望在 X 秒内未收到服务器响应时关闭连接，可以设置 TimeOut 间隔。

示例：每 30 秒发送一次 ping

```
oClient = new TsgcWebSocketClient();
oClient->HeartBeat->Interval = 30;
oClient->HeartBeat->Timeout = 0;
oClient->HeartBeat->Enabled = true;
oClient->Active = true;
```

有一个名为 **OnBeforeHeartBeat** 的事件，允许自定义 HeartBeat 行为。默认情况下，若启用了 HeartBeat，客户端将每隔 HeartBeat.Interval 属性设定的 X 秒发送一个 WebSocket ping。

OnBeforeHeartBeat 有一个名为 Handled 的参数，默认值为 false，表示流程由 TsgcWebSocketClient 组件控制。如果将该值设为 True，则不会发送 ping，您可以使用 Connection 类发送自定义消息。

WatchDog

如果启用了 WatchDog，当客户端检测到断开连接时，WatchDog 会每隔 X 秒尝试重新连接，直到连接恢复活跃。

示例：断开连接后每隔 10 秒重新连接一次，尝试次数不限。

```
oClient = new TsgcWebSocketClient();
oClient->WatchDog->Interval = 10;
oClient->WatchDog->Attempts = 0;
oClient->WatchDog->Enabled = true;
oClient->Active = true;
```

您可以使用 **OnBeforeWatchDog** 事件来更改客户端将尝试连接的服务器。**示例：**经过 3 次重试后，如果客户端无法连接到某台服务器，它将尝试连接到备用服务器。

Handled 属性若设为 True，表示客户端不会尝试重新连接。

TsgcWebSocketClient | 掉线断开

一旦建立连接，如果没有任何对等方发送任何数据，则不会在网络上发送任何数据包。TCP 是一种空闲协议，因此它假定连接仍然处于活动状态。

断开连接原因

- **应用程序关闭**：进程结束时，通常会发送 FIN 数据包，通知对端连接已关闭。但如果进程崩溃，则无法保证该数据包会被发送至对端。
- **设备关闭**：如果设备关闭，很可能不会有任何关于此的通知。
- **网线拔出**：如果网线被拔出，情况与路由器关闭相同；由于没有数据传输，连接不会被关闭。
- **路由器信号丢失**：如果应用程序失去路由器信号，连接仍将保持活跃。

检测半开放断连

您可以尝试使用以下方法 检测断开连接

第二个连接

您可以尝试打开第二个连接并尝试连接，但这有一些缺点：您会消耗更多资源，创建新线程等。此外，如果对端已重启，第二个连接可以正常工作，但第一个则不行。

Ping 另一个对等方

如果您尝试在半开连接上发送 ping 或任何消息，您会发现不会收到任何错误。

在 TCP 套接字级别启用 KeepAlive

TCP 保活数据包本质上是一个 ACK，其序列号比当前连接序列号小 1。收到此类 ACK 的主机将以当前序列号的 ACK 作为响应。保活可用于验证连接远端的计算机是否仍然可用。若在 TCP 连接上没有其他数据或更高层级的保活传输，TCP 保活可每隔 TCPKeepAlive.Time（默认为 7,200,000 毫秒即两小时）发送一次。若保活无响应，则每隔 TCPKeepAlive.Interval 秒重试一次。KeepAliveInterval 默认为 1000 毫秒。

您可以启用每个连接的 KeepAlive，让 TCP 协议检查连接是否处于活跃状态。如果您想检测断开的连接（例如拔掉网线时），这是首选方法。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
oClient->TCPKeepAlive->Enabled = true;
oClient->TCPKeepAlive->Time = 5000;
oClient->TCPKeepAlive->Interval = 1000;
```

TsgcWebSocketClient | 连接 TCP 服务器

TsgcWebSocketClient 可以连接到 WebSocket 服务器，但也可以连接到普通 TCP 服务器。

URL 属性

连接到 TCP 服务器最简单的方法是使用 **URL** 属性并将 **Active = true** 设置。

示例：连接到 127.0.0.1 端口 5555

```
oClient = new TsgcWebSocketClient();  
oClient->URL = "tcp://127.0.0.1:5555";  
oClient->Active = true;
```

主机、端口和参数

您可以使用 Host 和 port 属性连接到 TCP 服务器。

示例：连接到 127.0.0.1 端口 5555

```
oClient = new TsgcWebSocketClient();  
oClient->Specifications->RFC6455 = false;  
oClient->Host = "127.0.0.1";  
oClient->Port = 5555;  
oClient->Active = true;
```

TsgcWebSocketClient | 连接 TIME_WAIT

当客户端发起与服务器断开连接时，客户端和服务端之间会进行交换，以传达断开状态。流程完成后，客户端套接字连接进入 TIME_WAIT 状态，持续时间不固定。这是正常行为；在 Windows 操作系统中，此时间默认约为 4 分钟。

您可以减少或消除此行为，但请谨慎操作，可使用以下替代方案。

REGEDIT

您可以使用 Windows 注册表编辑器降低 TIME_WAIT 值

1. 打开注册表编辑器，访问 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\ Services\TCPIP\Parameters 注册表子键。
2. 创建名为 **TcpTimedWaitDelay** 的新 REG_DWORD 值
3. 设置秒数值。示例：如果您设置值为 5，则表示 TIME_WAIT 最多等待 5 秒。
4. 保存并重启系统。

LINGER

避免 TIME_WAIT 状态的另一个选项是使用套接字选项 SO_LINGER。若启用，客户端将重置连接而非正常关闭，从而避免 TIME_WAIT 状态。

您可以使用 **LingerState** 属性启用此选项，默认值为 -1。如果设置为零，则在断开套接字连接时不超时地重置连接。

此选项可能是最不推荐的，应仅作为最后手段使用。

TsgcWebSocketClient | WebSocket 重定向

当客户端连接到 WebSocket 服务器时，服务器可能返回 HTTP 响应码 30x。如果响应码为 301，表示位置已永久移动，新 URL 在 Location HTTP 头中提供。

WebSocket 客户端会自动处理重定向，因此如果检测到服务器响应包含重定向，它将断开当前连接并尝试连接到新的 Location URL。

示例

1. 客户端首先尝试连接到 URL `ws://127.0.0.1:5000`
2. 服务器返回响应代码 301，并在 Header Location 中包含值 `ws://80.50.1.2:3000`
3. 客户端从服务器读取响应，检测到这是一个重定向，并读取 Location 标头。
 1. 首先断开当前连接。
 2. 使用 Location 请求头的值 (`ws://80.50.1.2:3000`) 更新 URL 属性。
 3. 连接到新服务器。

TsgcWebSocketClient | 连接安全服务器

TsgcWebSocketClient 可以使用安全和非安全连接连接到 WebSocket 服务器。

您可以使用 URL 属性或 Host/Port 属性配置安全连接，请参阅[连接到 WebSocket 服务器](#)。

TLSOptions

在 **TLSOptions** 属性中，提供了自定义安全连接所需的各项属性。其中最重要的是 **version**，它指定 TLS 协议版本。对于绝大多数 WebSocket 服务器，通常只需将 **TLS** 属性设置为 **true**，并将 **TLSOptions.Version** 设置为 **tlsUndefined** 即可。

TLSOptions.Version 允许您设置连接服务器时使用的 TLS 版本，或让客户端从所有可用版本中协商 TLS 版本（当值为 **tlsUndefined** 时，这是默认行为）。

如果在尝试连接服务器时出现 TLS 协议错误，很可能是服务器需要比您所设置的更新的 TLS 版本。

如果 **TLSOptions.IOHandler** 设置为 **iohOpenSSL**，您需要部署 **OpenSSL 库**（这些库处理所有 TLS 相关事务），请查阅以下关于 [OpenSSL](#) 的文章。

如果 **TLSOptions.IOHandler** 设置为 **iohSChannel**，则无需部署任何库（仅支持 Windows）。

TsgcWebSocketClient | 证书 OpenSSL

当服务器要求客户端使用 SSL 证书连接时，使用 TsgcWebSocketClient 的 TLSOptions 属性设置证书文件。证书必须为 PEM 格式，如果证书格式不同，必须先将其转换为 PEM 格式。

通过 OpenSSL 库建立连接需要将 TLSOptions.IOHandler 设置为 iohOpenSSL。

配置以下属性：

- **CertFile**：PEM 格式证书的路径。
- **KeyFile**：证书私钥的路径。
- **RootCertFile**：证书根路径。
- **Password**：如果证书受密码保护，请在此处设置密钥。

TsgcWebSocketClient | SChannel 证书

当服务器要求客户端使用 SSL 证书连接时，请使用 TsgcWebSocketClient 的 TLSOptions 属性来设置证书文件。

通过 SChannel 连接需要将 TLSOptions.IOHandler 设置为 iohSChannel。

SChannel 支持两种证书认证类型：

1. 使用 **PFX 证书**
2. 设置 Windows 系统中已安装证书的**哈希证书**。

PFX 证书

PFX 证书是包含证书和私钥的文件，有时您的证书为 PEM 格式，使用前必须将其转换为 PFX 格式。使用以下 openssl 命令将 PEM 证书转换为 PFX

```
openssl pkcs12 -inkey certificate-pem.key -in certificate-pem.crt -export -out certificate.pfx
```

一旦证书为 PFX 格式，您只需部署证书并将 **TLSOptions.CertFile** 属性设置为其路径即可。

```
TLSOptions.IOHandler = iohSChannel
TLSOptions.CertFile = <certificate path>
TLSOptions.Password = <certificate optional password>
```

哈希证书

如果证书已安装在 Windows 证书存储中，您只需知道证书指纹并在 TLSOptions.SChannel_Options 属性中设置即可。

在 **PowerShell** 中查找证书的哈希值非常简单，只需在证书容器上运行 **dir** 命令即可。

```
dir cert:\localmachine\my
```

哈希是十六进制的**Thumbprint**（指纹）值。

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint                               Subject
-----
C12A8FC8AE668F866B48F23E753C93D357E9BE10  CN=*.mydomain.com
```

获得指纹值后，您必须在 **TLSOptions.SChannel_Options** 属性中设置哈希值和证书位置。

```
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 获取连接信息

客户端连接到安全服务器后，您可以请求获取所使用的 TLS 版本（TLS 1.2、TLS 1.3 等）、所用密码套件、强度等相关信息。

调用 SChannel 处理程序的 **GetInfo** 函数以访问此信息。您可以使用 **OnSSLAfterCreateHandler** 方法访问 SSL 处理程序，该方法在 SChannel 处理程序创建后调用。客户端连接到服务器后，如果已分配 SSL 处理程序，请调用 **GetInfo** 函数，成功后将返回连接数据。

```
oClient = new TsgcWebSocketClient();
oClient->URL = "wss://www.esegece.com:2053";
oClient->TLSOptions->Version = tls1_2;
oClient->TLSOptions->IOHandler = iohSChannel;
oClient->OnSSLAfterCreateHandler = OnSSLAfterCreateHandlerEvent;
oClient->OnConnect = OnConnectEvent;
oClient->Active = true;
void OnSSLAfterCreateHandlerEvent(TObject *Sender, TwSSSLHandler aType,
    TIdSSLIOHandlerSocketBase *aSSLHandler)
{
    if (aSSLHandler->ClassType() == __classid(TsgcIdSSLIOHandlerSocketSChannel))
    {
        SSL = dynamic_cast<TsgcIdSSLIOHandlerSocketSChannel*>(aSSLHandler);
    }
}
void OnConnectEvent(TsgcWSConnection *Connection)
{
    if (SSL != NULL)
    {
        TsgcSChannelConnectionInfo oInfo = SSL->GetInfo();
        if (oInfo->Protocol != tls1_2)
        {
            throw Exception("Client cannot connect using TLS 1.2");
        }
    }
}
```

TsgcWebSocketClient | 客户端发送文本消息

客户端连接到服务器后，即可发送文本消息。要发送文本消息，只需调用 `WriteData()` 方法。

发送文本消息

调用 **`WriteData()`** 方法发送文本消息。此方法在调用它的**同一线程**上执行。

```
TsgcWebSocketClient1->WriteData("My First sgcWebSockets Message!.");
```

如果 **`QueueOptions.Text`** 的值不同于 **`qmNone`**，则它不会在被调用的同一线程上处理，而是在辅助线程上处理。默认情况下此选项处于禁用状态。

发送文本消息并等待响应

调用 **`WriteAndWaitData()`** 方法发送文本消息并等待服务器响应。该函数返回收到的文本消息。

```
TsgcWebSocketClient1->WriteAndWaitData("My First sgcWebSockets Message!.");
```

TsgcWebSocketClient | 客户端发送二进制消息

客户端连接到服务器后，可以发送二进制消息。要发送二进制消息，只需调用 `WriteData()` 方法。

发送二进制消息

调用 **`WriteData()`** 方法发送二进制消息。此方法在调用它的**同一线程**上执行。

```
TMemoryStream oStream = new TMemoryStream();  
...  
TsgcWebSocketClient1->WriteData(oStream);  
delete oStream;
```

若 **`QueueOptions.Binary`** 的值与 **`qmNone`** 不同，则不在调用所在的同一线程上处理，而是在辅助线程上处理。默认情况下，此选项已禁用。

发送二进制消息并等待响应

调用 `WriteAndWaitData()` 方法发送二进制消息并等待服务器响应。该函数返回接收到的二进制消息。

```
TsgcWebSocketClient1->WriteAndWaitDataData(oStream);
```

TsgcWebSocketClient | 客户端发送文本和二进制消息

WebSocket 协议只允许两种类型的消息：文本或二进制。但是，您不能在同一条消息中同时发送二进制数据和文本。

解决此问题的一种方法是在发送二进制消息之前向其添加头部，并在接收时解码该二进制消息。

sgcWebSocket_Helpers 中有 2 个函数可用于为二进制数据包设置简短描述。这实际上是向流添加一个头部，用于标识二进制数据包。

在发送二进制消息之前，调用该方法对流进行编码。

```
sgcWSStreamWrite("00001", oStream);  
TsgcWebSocketClient1->WriteData(oStream);
```

当收到二进制消息时，调用该方法解码数据流。

```
sgcWSStreamRead(oStream, vID);
```

唯一的限制是用于标识二进制消息的文本最大长度为 10 个字符（如果您有访问源代码的权限，可以修改此限制）。

TsgcWebSocketClient | 接收文本消息

当客户端收到文本消息时，将触发 **OnMessage** 事件。读取 **Text** 参数以获取收到的消息字符串。

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

默认情况下，客户端使用 **neAsynchronous** 方法分发 **OnMessage** 事件，这意味着此事件在主线程的上下文中执行，因此更新表单中的任何控件都是线程安全的。

如果您的客户端接收大量消息，或者需要控制与其他线程的同步，请将 **NotifyEvents** 属性设置为 **neNoSync**，这意味着 **OnMessage** 事件将在连接线程的上下文中执行，因此如果您需要更新窗体上的任何控件或访问共享对象，则必须实现自己的同步方法。

TsgcWebSocketClient | 接收二进制消息

当客户端收到二进制消息时，将触发 **OnBinary** 事件。读取 **Data** 参数以获取收到的二进制消息。

```
void OnBinary(TsgcWSConnection *Connection, const TMemoryStream *Data)
{
    oBitmap = new TBitmap();
    oBitmap->LoadFromStream(Data);
    Image1->Picture->Assign(oBitmap);
    Log(
        "#image uncompressed size: " + IntToStr(Data->Size) +
        ". Total received: " + IntToStr(Connection->RecBytes));
    delete oBitmap;
}
```

默认情况下，客户端使用 **neAsynchronous** 方法分发 **OnMessage** 事件，这意味着**此事件在主线程上下文中执行**，因此更新表单控件等操作是线程安全的。

如果你的客户端 接收大量消息，或者你需要控制与 其他线程的同步，请将 **NotifyEvents** 属性设置为 **neNoSync**，这意味着 **OnMessage** 事件将在**连接线程的上下文中执行**，因此如果你需要更新 某个窗体的控件或访问共享对象，你必须实现自己的同步方法。

TsgcWebSocketClient | 客户端身份验证

TsgcWebSocket 客户端支持 4 种认证类型：

- **Basic**：在 WebSocket 握手期间发送包含用户名和密码（以 Basic 身份验证编码）的 HTTP 标头。
- **Token**：在 WebSocket 握手期间以 HTTP 请求头的形式发送令牌。按服务器要求在 Authentication.Token.AuthToken 中设置所需令牌。
- **会话**：客户端首先向服务器请求 HTTP 会话，若服务器返回会话，则该会话将通过 WebSocket 握手的 GET HTTP 头传递。（* 这是 sgcWebSockets 库自有的认证方式。）
- **URL**：客户端使用 WebSocket 握手的 GET HTTP 标头请求授权。（* sgcWebSockets 库自有的身份验证方法）

Basic 认证

这是一种简单的授权方法，其中用户名和密码经过编码后作为 HTTP 头传递。只需设置 User 和 Password，并仅启用 Basic 授权类型即可使用此方法。

```
oClient = new TsgcWebSocketClient();  
oClient->
```

Authentication

```
->Enabled = true;  
oClient->
```

Authentication

```
->Basic->Enabled = true;  
oClient->
```

Authentication

```
->User = "your user";  
oClient->
```

Authentication

```
->Password = "your password";  
oClient->
```

Authentication

```
->Token->Enabled = false;  
oClient->
```

Authentication

```
->URL->Enabled = false;  
oClient->
```

Authentication

```
->Session->Enabled = false;  
oClient->Active = true;
```

认证令牌

允许您使用 JWT 进行授权。这需要您使用外部工具（例如 HTTP 连接、OAuth2 等）获取令牌。如果附加了 OAuth2 组件，即可自动获取该令牌。阅读更多关于 [OAuth2](#) 的内容。您必须设置 AuthToken 并启用令牌身份验证。

```
oClient = new TsgcWebSocketClient();  
oClient->
```

Authentication

```
->Enabled = true;  
oClient->
```

Authentication

```
->Token->Enabled = true;
oClient->
```

Authentication

```
->Token->AuthToken = "your token";
oClient->
```

Authentication

```
->Basic->Enabled = false;
oClient->
```

Authentication

```
->URL->Enabled = false;
oClient->
```

Authentication

```
->Session->Enabled = false;
oClient->Active = true;
```

认证会话

客户端首先使用 HTTP 连接连接到服务器，请求一个新的会话。成功后，服务器返回一个 SessionId，客户端在 WebSocket 握手的 GET HTTP 头部中发送该 SessionId。需要设置 Username 和 Password 并启用会话身份验证。

```
oClient = new TsgcWebSocketClient();
oClient->
```

Authentication

```
->Enabled = true;
oClient->
```

Authentication

```
->Session->Enabled = true;
oClient->
```

Authentication

```
->User = "your user";
oClient->
```

Authentication

```
->Password = "your password";
oClient->
```

Authentication

```
->Basic->Enabled = false;
oClient->
```

Authentication

```
->URL->Enabled = false;
oClient->
```

Authentication

```
->Token->Enabled = false;
oClient->Active = true;
```

身份验证 URL

此身份验证方法在 WebSocket 握手的 GET HTTP 请求头中传递用户名和密码。

```
oClient = new TsgcWebSocketClient();
oClient->
```

Authentication

```
->Enabled = true;
oClient->
```

Authentication

```
->URL->Enabled = true;  
oClient->
```

Authentication

```
->User = "your user";  
oClient->
```

Authentication

```
->Password = "your password";  
oClient->
```

Authentication

```
->Basic->Enabled = false;  
oClient->
```

Authentication

```
->Session->Enabled = false;  
oClient->
```

Authentication

```
->Token->Enabled = false;  
oClient->Active = true;
```

TsgcWebSocketClient | 客户端异常

有时通信会出现错误：服务器可能因连接未授权或消息格式不正确而断开连接。有 2 个事件用于捕获错误。

OnError

每次 WebSocket 协议发生错误时触发此事件，例如无效的消息类型、无效的 utf8 字符串.....

```
void OnError(TsgcWSConnection *Connection, const string Error)
{
    WriteLn("#error: " + Error);
}
```

OnException

每次发生异常时触发此事件，例如向未激活的套接字写入数据或访问不存在的对象。

```
void OnException(TsgcWSConnection *Connection, Exception *E)
{
    WriteLn("#exception: " + E->Message);
}
```

默认情况下，当连接被服务器关闭时，将触发异常。若不希望触发这些异常，请禁用 **Options.RaiseDisconnectExceptions**。

TsgcWebSocketClient | WebSocket 握手

WebSocket 协议使用 HTTP 握手将 HTTP 协议升级为 WebSocket 协议。此握手由 TsgcWebSocket 客户端组件内部处理，但若服务器需要额外的 HTTP 头信息，您可以添加自定义 HTTP 请求头。

示例：如果您需要添加 HTTP 请求头 "Client: sgcWebSockets"

```
void OnHandshake(TsgcWSConnection *Connection, ref TStringList *Headers)
{
    Headers->Add("Client: sgcWebSockets");
}
```

TsgcWebSocketClient | 客户端注册协议

默认情况下，TsgcWebSocketClient 不使用任何 SubProtocol。WebSocket 子协议建立在 WebSocket 协议之上，定义了自定义消息协议。WebSocket 子协议的例子包括 MQTT、STOMP 等。

WebSocket 子协议名称作为 HTTP 头在 WebSocket 握手中发送。服务器处理此头，如果服务器支持该子协议，将接受连接。如果不支持，连接将自动关闭。

示例： 连接到子协议名称为 'myprotocol' 的 WebSocket 服务器

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


TsgcWebSocketClient | 客户端代理

TsgcWebSocket 客户端支持通过代理进行连接。要配置 代理连接，只需填写 TsgcWebSocket 客户端的 **Proxy** 属性即可。

```
Client = new TsgcWebSocketClient();
Client->Proxy->Enabled = true;
Client->Proxy->Username = "user";
Client->Proxy->Password = "secret";
Client->Proxy->Host = "80.55.44.12";
Client->Proxy->Port = 8080;
Client->Active = true;
```

TsgcTCPClient

TsgcTCPClient 是一个 TCP 客户端组件，允许连接到普通 TCP 服务器。

简介

TsgcTCPClient 是一个 TCP 客户端组件，允许连接到普通 TCP 服务器。支持 TLS/SSL 连接、代理服务器、心跳监控、自动重连（WatchDog）和 TCP 保活。

1. 将 **TsgcTCPClient** 组件拖放到窗体上。
2. 设置 **Host** 和 **Port**（默认为 80）以连接到可用的 TCP 服务器。
3. 若需要安全连接，请将 **TLS** 设置为 True（默认值为 False）。
4. 如果通过代理服务器连接，可选择配置 **Proxy**。
5. 处理事件：**OnConnect**、**OnDisconnect**、**OnMessage**、**OnBinary**、**OnError**、**OnException**。
6. 将 **Active** 设置为 True 以打开连接。

参考

- 属性——组件上共有 23 个可用属性。
- 方法 — 7 个公共方法。
- 事件 — 12 个可处理的事件。

TsgcTCPClient

TsgcTCPClient 是一个 TCP 客户端组件，允许连接到普通 TCP 服务器。

简介

TsgcTCPClient 是一个 TCP 客户端组件，允许连接到普通 TCP 服务器。支持 TLS/SSL 连接、代理服务器、心跳监控、自动重连（WatchDog）和 TCP 保活。

1. 将 **TsgcTCPClient** 组件拖放到窗体上。
2. 设置 **Host** 和 **Port**（默认为 80）以连接到可用的 TCP 服务器。
3. 若需要安全连接，请将 **TLS** 设置为 True（默认值为 False）。
4. 如果通过代理服务器连接，可选择配置 **Proxy**。
5. 处理事件：**OnConnect**、**OnDisconnect**、**OnMessage**、**OnBinary**、**OnError**、**OnException**。
6. 将 **Active** 设置为 True 以打开连接。

参考

- 属性——组件上共有 23 个可用属性。
- 方法 — 7 个公共方法。
- 事件 — 12 个可处理的事件。

TsgcWebSocketServer

TsgcWebSocketServer 实现了服务器 WebSocket 组件，可以处理多线程客户端连接。

简介

TsgcWebSocketServer 实现了服务器 WebSocket 组件，可处理多个线程化客户端连接。请按照以下步骤配置此组件：

1. 将 TsgcWebSocketServer 组件拖放到窗体上。
2. 设置端口（默认为 80）。如果您在防火墙后面，可能需要配置防火墙。
3. 设置允许的规范，默认情况下允许所有规范。

RFC6455：是标准且推荐的 WebSocket 规范。

Hixie76：这是一个草案版本，建议仅在需要支持 Safari 4.2 等旧版浏览器时才建立 Hixie76 连接。

```
void __fastcall TForm1::OnTCPConnectEvent(TsgcWSConnection *aConnection, bool &Accept)
{
    aConnection->Transport = trpTCP;
    Accept = true;
}
```

5. 创建一个过程并将属性 Active 设置为 True。

参考

- 属性 — 组件上提供 27 个属性。
- Methods — 7 个公共方法。
- 事件 — 22 个可处理的事件。

指南

- 启动
 - [服务器启动](#)
 - [服务器绑定](#)
 - [服务器启动与关闭](#)
 - [服务器保持活跃](#)
- 连接
 - [服务器保持连接活跃](#)
 - [服务器纯 TCP](#)
 - [服务器关闭连接](#)
 - [客户端连接](#)
- 身份验证
 - [服务器身份验证](#)

- 发送消息
 - [服务器发送文本消息](#)
 - [服务器发送二进制消息](#)
- 接收消息
 - [服务器接收文本消息](#)
 - [服务器接收二进制消息](#)
- SSL
 - [服务器 SSL](#)
 - [服务器 SSL SChannel](#)

TsgcWebSocketServer | 启动服务器

启动服务器时，首先必须设置监听端口。默认情况下设置为端口 80，但您可以将其更改为任意端口。端口设置完成后，有两种方法可以启动服务器。

Active 属性

如果将 Active 属性设置为 true，服务器将开始在配置的端口上监听所有传入连接。

```
oServer = new TsgcWebSocketServer();  
oServer->Port = 80;  
oServer->Active = true;
```

如果将 Active 属性设置为 false，服务器将停止并关闭所有活跃连接。

```
oServer->Active = false;
```

Start / Stop 方法

设置 Active 属性会在同一线程中启动/停止服务器，而 Start 和 Stop 方法则在辅助线程中执行。

```
oServer = new TsgcWebSocketServer();  
oServer->Port = 80;  
oServer->Start();
```

如果您调用 Stop() 方法，服务器将停止并关闭所有活动连接。

```
oServer->Stop();
```

您可以使用 **ReStart** 方法，在辅助线程中停止并启动服务器。

如果在关闭服务器后更改了端口以监听不同端口，请在关闭服务器后调用 **Bindings.Clear()** 方法删除所有旧的绑定。否则，服务器将尝试绑定到旧的绑定地址。

TsgcWebSocketServer | 服务器绑定

默认情况下，如果您只填写 **Port** 属性，服务器将在所有 IP 上绑定监听端口，例如，如果您有 3 个 IP：127.0.0.1、80.54.11.22 和 12.55.41.17，您的服务器将在这 3 个 IP 上绑定此端口。
通常建议仅绑定到所需的 IP。这就是可以使用 Bindings 属性的地方。
无需使用 Port 属性，直接使用 Bindings 属性并填写所需的 IP 和端口即可。

示例：将端口 5555 绑定到 IP 127.0.0.1 和 IP 80.58.25.40

```
oServer = new TsgcWebSocketServer();  
binding = oServer->Bindings->Add();  
binding->IP = "127.0.0.1";  
binding->Port = 5555;  
binding = oServer->Bindings->Add();  
binding->IP = "80.58.25.40";  
binding->Port = 5555;  
oServer->Active = true;
```

如果在关闭服务器后更改端口，以便在不同端口上开始监听，请在关闭服务器后调用方法 **Bindings.Clear()** 以删除所有之前的绑定。否则服务器将尝试绑定到之前的绑定。

TsgcWebSocketServer | 服务器启动与停止

设置好服务器的所有必要配置后，有 2 个有用的事件可以知道服务器何时启动以及何时停止。

OnStartup

此事件在服务器启动并可以处理新连接时触发。

```
void OnStartup(TObject *Sender)
{
    WriteLn("#server started");
}
```

OnShutdown

此事件在服务器停止且不再接受连接后触发。

```
void OnShutdown(TObject *Sender)
{
    WriteLn("#server stopped");
}
```


TsgcWebSocketServer | 保持服务器活跃

服务器启动后，有时可能因任何原因停止。如果您希望在意外停止后重启服务器，可以使用 WatchDog 属性。

WatchDog

如果启用了 WatchDog，当服务器检测到关闭时，WatchDog 会每隔 X 秒尝试重启，直到服务器再次活跃。

示例：意外停止后每 10 秒重启一次，不限重试次数。

```
oServer = new TsgcWebSocketServer();  
oServer->WatchDog->Interval = 10;  
oServer->WatchDog->Attempts = 0;  
oServer->WatchDog->Enabled = true;  
oServer->Active = true;
```

TsgcWebSocketServer | 服务器 SSL

服务器可以配置为使用 **SSL 证书**。要获得带有服务器证书的生产服务器，您必须购买来自知名提供商（Namecheap、GoDaddy、Thawte 等）的证书。出于**测试目的**，您可以使用**自签名证书**（查看使用自签名证书的 Demos/Chat 示例）。

证书必须为 **PEM 格式**，PEM（隐私增强邮件）定义于 RFC 1421 至 1424，是一种容器格式，可以仅包含公共证书（如 Apache 安装中的 CA 证书文件 /etc/ssl/certs），也可以包含完整的证书链（包括公钥、私钥和根证书）。要创建单个 PEM 证书，只需打开私钥文件，复制其内容并粘贴到证书文件中。

示例：

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-----
```

要启用 SSL，只需 **启用 SSL 属性** 并配置 **CertFile**、**KeyFile** 和 **RootFile** 的路径。如果证书包含完整的证书链（公钥、私钥等），只需将所有路径指向同一证书即可。

另一个必须设置的属性是 **SSLOptions.Port**，这是用于安全连接的端口。

简单 SSL 配置

示例：在 IP 127.0.0.1 和端口 443 上配置 SSL

```
oServer = new TsgcWebSocketServer();
oServer->SSL = true;
oServer->SSLOptions->CertFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->KeyFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->RootCertFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->Active = true;
```

SSL 和非 SSL

您可以将服务器配置为在多个 IP 和端口上监听；请参阅 [绑定说明文章](#)，了解其工作原理。服务器可同时支持 SSL 和非 SSL 连接（当然需要在不同端口上监听）。您只需绑定到两个不同的端口，并将一个端口配置为 SSL 连接，另一个配置为非 SSL 连接。

示例：在 IP 127.0.0.1、端口 80（非加密）和 443（SSL）上配置服务器

```
oServer = new TsgcWebSocketServer();
bind = oServer->Bindings->Add();
bind->IP = "127.0.0.1";
bind->Port = 80;
bind = oServer->Bindings->Add();
bind->IP = "127.0.0.1";
bind->Port = 443;
oServer->Port = 80;
oServer->SSL = true;
oServer->SSLOptions->CertFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->KeyFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->RootCertFile = "c:\\certificates\\mycert.pem";
oServer->SSLOptions->Port = 443;
oServer->Active = true;
```

服务器 SSL | Indy 服务器的 SChannel

基于 Indy 的服务器组件 (**TsgcWebSocketServer**、**TsgcWebSocketHTTPServer**) 可以使用 **Windows SChannel** (安全通道) 作为 TLS 提供程序, 而非 OpenSSL。SChannel 是 Windows 原生的 TLS 实现, 无需任何外部 DLL。

工作原理

当服务器组件启用了 SSL 且 IOHandler 设置为 **iohSChannel** 时, 服务器会创建一个 **TsgcServerIOHandlerSSLSChannel** 实例, 使用 Windows SChannel API 处理所有 TLS 操作。对于每一个传入的客户端连接, 服务器通过 SChannel 提供程序执行 TLS 握手, 协商协议版本、密码套件, 并绑定已配置的证书。SChannel 从 **Windows 证书存储** 或 **PFX 文件** (.pfx / .p12) 读取证书。无需 PEM 文件, 也无需部署 OpenSSL 库。

配置

要在 Indy 服务器上启用 SChannel, 请配置以下属性:

1. 将 **SSL** 属性设置为 **True**。
2. 将 **SSLOptions.IOHandler** 设置为 **iohSChannel**。
3. 将 **SSLOptions.Version** 设置为所需的 TLS 版本 (tls1_2、tls1_3 等)。
4. 将 **SSLOptions.Port** 设置为用于安全连接的端口。
5. 使用下面描述的两种方法之一配置证书。

SChannel_Options 属性

SSLOptions.SChannel_Options 子属性公开了 SChannel 专用设置:

属性	描述
CertHash	安装在 Windows 证书存储中的证书指纹 (十六进制哈希值)。
CertStoreName	要搜索的证书存储: scsnMY (个人)、scsnRoot、scsnTrust、scsnCA。
CertStorePath	存储位置: scspStoreLocalMachine 或 scspStoreCurrentUser。
CipherList	可选的冒号分隔加密算法列表 (例如 CALG_AES_256:CALG_AES_128)。留空则使用系统默认值。
UseLegacyCredentials	当为 True 时, 使用旧版 SCHANNEL_CRED 结构代替 SCH_CREDENTIALS。对于不支持更新 API 的旧版 Windows, 请启用此选项。

此外, 通用 **SSLOptions** 属性 **CertFile** 和 **Password** 在从 PFX 文件加载证书时使用。

来自 Windows 存储的证书

如果证书已安装在 Windows 证书存储中, 请提供证书 **指纹** 并指明其所在位置。要查找指纹, 请打开 **PowerShell** 并运行:

```
dir cert:\localmachine\my
```

Thumbprint 列显示您所需的十六进制哈希值。

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint Subject
-----
C12A8FC8AE668F866B48F23E753C93D357E9BE10 CN=*.mydomain.com
```

```
oServer = new TsgcWebSocketServer();
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->Version = tls1_2;
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->SSLOptions->SChannel_Options->CertHash = "C12A8FC8AE668F866B48F23E753C93D357E9BE10";
oServer->SSLOptions->SChannel_Options->CertStoreName = scsnMY;
oServer->SSLOptions->SChannel_Options->CertStorePath = scspStoreLocalMachine;
oServer->Active = true;
```

来自 PFX 文件的证书

如果您有 PFX (.pfx 或 .p12) 证书文件，请在 SSLOptions 上设置 **CertFile** 和 **Password** 属性。SChannel 将在启动时导入证书。

```
oServer = new TsgcWebSocketServer();
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->Version = tls1_2;
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->SSLOptions->CertFile = "c:\\certificates\\server.pfx";
oServer->SSLOptions->Password = "mypassword";
oServer->Active = true;
```

如果您有 PEM 证书和私钥，请先使用 OpenSSL 将其转换为 PFX 格式：

```
openssl pkcs12 -inkey server.key -in server.crt -export -out server.pfx
```

TLS 版本

使用 **SSLOptions.Version** 属性控制服务器接受的 TLS 版本：

值	描述
tls1_0	TLS 1.0（不推荐）
tls1_1	TLS 1.1
tls1_2	TLS 1.2（推荐）
tls1_3	TLS 1.3
tlsUndefined	接受 TLS 1.0、1.1 和 1.2

密码套件列表

默认情况下，SChannel 使用系统密码配置。您可以通过将 **SChannel_Options.CipherList** 设置为以冒号分隔的算法名称列表来限制允许的密码，例如：

```
CALG_AES_256:CALG_AES_128
```

将此属性留空以使用 Windows 默认值。

旧版凭据

Windows Server 2019 及更早版本可能不支持较新的 **SCH_CREDENTIALS** API。如果服务器无法在较旧的 Windows 版本上启动，请将 **SChannel_Options.UseLegacyCredentials** 设置为 **True** 以改用旧版 **SCHANNEL_CRED** 结构。

该组件在大多数情况下会自动检测 Windows 版本，但如有需要，您也可以强制启用传统模式。

相对于 OpenSSL 的优势

- **无需外部 DLL**：SChannel 已内置于 Windows，因此无需部署 libeay32.dll / ssleay32.dll 或 libcrypto / libssl。
- **Windows 证书存储集成**：使用操作系统已安装和管理的证书。
- **自动更新**：TLS 改进和安全补丁通过 Windows Update 应用。

注意事项

- SChannel 仅在 **Windows** 上可用。对于跨平台服务器，请使用 OpenSSL（iohOpenSSL）。
- 服务器必须拥有与证书关联的**私钥**。使用 Windows 存储方法时，证书必须连同私钥一起导入。
- 对于使用证书存储方式的生产服务器，推荐使用 **本地计算机** 存储 (scspStoreLocalMachine)，以确保无论服务以哪个用户账户运行，证书均可访问。
- 仅支持 **PFX** (.pfx/.p12) 证书文件。如果您有 PEM 文件，请先将其转换为 PFX 格式。

TsgcWebSocketServer | 服务器验证证书

默认情况下，服务器不验证对端证书。要配置服务器验证客户端证书，请执行以下步骤：

1. 将属性 `SSLOptions.VerifyCertificate` 设置为 `true`

处理 `OnSSLVerifyPeer` 事件并实现以下代码，以便在每次客户端使用证书连接时收到通知。

```
void OnSSLSSLVerifyPeer(System::TObject* Sender, TIdX509* Certificate, bool &Accept)
{
    // ... validate the certificate
    if (Certificate_OK)
        Accept = true;
    else
        Accept = false;
}
```

请注意，`OnSSLVerifyPeer` 事件仅在客户端提供证书时调用，若客户端未提供证书，则不会触发该事件。

您可以将服务器配置为仅允许使用证书的 **SSL 连接**。为此，请设置以下属性：

- `SSLOptions.VerifyCertificate_Options.FailIfNoCertificate = true`

如果客户端未提供证书，连接将在 SSL 握手时关闭。

TsgcWebSocketServer | 服务器保持连接活跃

客户端连接到服务器后，有时由于信号差、连接错误等原因连接可能会断开。使用 HeartBeat 保持连接活跃。

HeartBeat

HeartBeat 属性允许您每隔 X 秒发送一个 **Ping**，以保持连接活跃。某些客户端会在对端之间没有数据交换时关闭 TCP 连接。HeartBeat 通过在特定间隔发送 ping 来解决此问题。通常这足以维持连接活跃，但如果希望在 X 秒内未收到客户端响应时关闭连接，可以设置 TimeOut 间隔。

示例：每 30 秒向所有已连接的客户端发送一次 ping

```
oServer = new TsgcWebSocketServer();
oServer->HeartBeat->Interval = 30;
oServer->HeartBeat->Timeout = 0;
oServer->HeartBeat->Enabled = true;
oServer->Active = true;
```


TsgcWebSocketServer | 服务器纯 TCP

WebSocket 服务器接受 WebSocket、HTTP、SSE 及其他协议，但也可与纯 TCP 连接配合使用。请阅读关于 [TCP 连接](#) 的详细信息。

有 2 个事件可用于处理 TCP 连接。

OnTCPConnect

此事件在客户端连接到服务器之后、客户端与服务器之间任何握手之前调用。OnConnect 事件仅在客户端发送消息后触发（以允许服务器检测正在使用的协议）。

此事件允许您了解是否有新客户端正在尝试连接到服务器，服务器可以接受或拒绝该连接。默认情况下，服务器始终接受连接。

OnUnknownProtocol

当服务器从客户端接收到第一条消息但无法检测到它是任何已知协议时，将调用此事件。在此事件中，服务器可以接受或拒绝该协议。

OnConnect

此事件在成功且完整的连接后触发。如果连接是普通 TCP，则在 OnUnknownProtocol 事件中接受协议后触发。

TsgcWebSocketServer | 服务器关闭连接

单个连接可以使用 Close 或 Disconnect 方法关闭。

断开连接

[TsgcWSConnection](#) 有一个名为 Disconnect() 的方法，允许您在套接字层面断开连接。调用此方法后，套接字将直接断开，不会等待客户端的任何响应。您可以通过此方法发送关闭码。

关闭

[TsgcWSConnection](#) 有一个名为 Close() 的方法，允许您向客户端发送请求关闭连接的消息。客户端收到此消息后，必须关闭连接，服务器将收到连接已关闭的通知。您可以使用此方法发送关闭代码。

DisconnectAll

断开所有活跃连接。此方法在服务器停止监听之前自动调用，但您也可以随时调用此方法。

TsgcWebSocketServer | 客户端连接

要访问活跃的客户端连接，可以使用 Connections 属性遍历列表并访问客户端连接类。Connections 属性访问一个线程安全列表，因此首先锁定列表，完成后解锁列表。

```
void DoClientIPAddresses() {
    TList *oList;
    TsgcWSConnectionServer *oConnection;
    oList = TsgcWebSocketHTTPServer1->LockList();
    try {
        for (int i = 0; i < oList->Count; ++i) {
            oConnection = dynamic_cast<TsgcWSConnectionServer*>(static_cast<TIdContext*>(oList->Items[i])->Data);
            ShowMessage(oConnection->IP + ":" + IntToStr(oConnection->Port));
        }
    } __finally {
        TsgcWebSocketHTTPServer1->UnlockList();
    }
}
```

TsgcWebSocketServer | 服务器身份验证

TsgcWebSocket 服务器支持 3 种类型的身份验证：

- **Basic**：在 WebSocket 握手期间读取 HTTP 头部，其中用户名和密码以 Basic Authorization 方式编码。
- **Session**：客户端首先从服务器请求 HTTP 会话，如果服务器返回会话，它将在 WebSocket 握手的 GET HTTP 标头中传递。（* sgcWebSockets 库的自有授权方法）
- **URL**：使用 WebSocket 握手的 GET HTTP 头读取请求授权。（* sgcWebSockets 库的自有授权方式）。

您可以使用 **AuthUsers** 属性设置已认证用户列表，只需按以下格式设置用户：user=password

OnAuthentication

每当服务器从客户端收到身份验证请求时，都会调用此事件，以指示用户是否已通过身份验证。使用 **Authenticated** 参数来接受或拒绝连接。

```
void OnAuthentication(TsgcWSConnection *Connection, string aUser, string aPassword,
    ref bool Authenticated)
{
    if ((aUser == "user") && (aPassword == "secret"))
    {
        Authenticated = true;
    }
    else
    {
        Authenticated = false;
    }
}
```

OnUnknownAuthentication

如果身份验证类型不受默认支持（例如 JWT），您仍可使用此事件来接受或拒绝连接。只需读取参数并决定是否接受连接即可。

```
void OnUnknownAuthentication(TsgcWSConnection *Connection, string AuthType, string AuthData,
    ref string aUser, ref string aPassword, ref bool Authenticated)
{
    if (AuthType == "Bearer")
    {
        if (AuthData == "jwt_token")
        {
            Authenticated = true;
        }
        else
        {
            Authenticated = false;
        }
    }
    else
    {
        Authenticated = false;
    }
}
```

TsgcWebSocketServer | 服务器发送文本消息

客户端连接到服务器后，服务器可以发送文本消息。要发送文本消息，请调用 `WriteData()` 方法向 单个客户端发送消息，或使用 `Broadcast` 向所有客户端发送消息。

发送文本消息

调用 `WriteData()` 方法以 发送文本消息。

```
TsgcWebSocketServer1->WriteData("guid", "My First sgcWebSockets Message!.");
```

如果 `QueueOptions.Text` 的值不同于 `qmNone`，则它不会在被调用的同一线程上处理，而是在辅助线程上处理。默认情况下此选项处于禁用状态。

当 `IOHandlerOptions.IOHandlerType = iohIOCP` 时，`QueueOptions` 不起作用（由于 IOCP 架构，不支持此功能）。

您也可以从 `TsgcWSConnection` 调用 `WriteData()` 方法，**示例**：在客户端连接到服务器时向其发送消息。

```
void OnConnect(TsgcWSConnection *Connection)
{
    Connection->WriteData("Hello From Server");
}
```

向所有已连接的客户端发送消息

调用 `Broadcast()` 方法向所有已连接的客户端发送文本消息。

```
TsgcWebSocketServer1->Broadcast("Hello From Server");
```

TsgcWebSocketServer | 服务器发送二进制消息

客户端连接到服务器后，服务器可以发送二进制消息。调用 `WriteData()` 方法向单个客户端发送消息，或使用 `Broadcast` 向所有客户端发送消息。

发送二进制消息

调用 **`WriteData()`** 方法发送二进制消息。

```
TsgcWebSocketServer1->WriteData("guid", new TMemoryStream());
```

如果 **`QueueOptions.Binary`** 的值不同于 **`qmNone`**，则将在辅助线程上处理，而不是在调用所在的同一线程上处理。默认情况下此选项禁用。

当属性 **`IOHandlerOptions.IOHandlerType = iohIOCP`** 时，**`QueueOptions`** 不起作用（由于 IOCP 架构，该功能不受支持）。

您也可以从 **`TsgcWSConnection`** 调用 `WriteData()` 方法，**示例**：在客户端连接到服务器时向其发送消息。

```
void OnConnect(TsgcWSConnection *Connection)
{
    Connection->WriteData(new TMemoryStream());
}
```

向所有已连接的客户端发送消息

调用 **`Broadcast()`** 方法向所有已连接的客户端发送二进制消息。

```
TsgcWebSocketServer1->Broadcast(new TMemoryStream());
```

TsgcWebSocketServer | 服务器接收文本消息

当服务器收到文本消息时，触发 **OnMessage** 事件。读取 **Text** 参数以获取接收到的消息字符串。

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

默认情况下，服务器使用 **neAsynchronous** 方式分发 **OnMessage** 事件，这意味着该事件在主线程上下文中执行，因此更新表单控件等操作是线程安全的。

如果您的服务器接收大量消息或需要控制与其他线程的同步，请将 **NotifyEvents** 属性设置为 **neNoSync**，这意味着 **OnMessage** 事件将在连接线程的上下文中执行，因此如果您需要更新窗体的任何控件或访问共享对象，必须实现自己的同步方法。

TsgcWebSocketServer | 服务器接收二进制消息

当服务器收到二进制消息时，将触发 **OnBinary** 事件。读取 **Data** 参数以获取收到的二进制消息。

```
void OnBinary(TsgcWSConnection *Connection, const TMemoryStream *Data)
{
    oBitmap = new TBitmap();
    oBitmap->LoadFromStream(Data);
    Image1->Picture->Assign(oBitmap);
    Log(
        "#image uncompressed size: " + IntToStr(Data->Size) +
        ". Total received: " + IntToStr(Connection->RecBytes));
    oBitmap.Free();
}
```

默认情况下，服务器使用 **neAsynchronous** 方式分发 **OnMessage** 事件，这意味着该事件在主线程上下文中执行，因此更新表单控件等操作是线程安全的。

如果您的服务器接收大量消息或需要控制与其他线程的同步，请将 **NotifyEvents** 属性设置为 **neNoSync**，这意味着 **OnMessage** 事件将在连接线程的上下文中执行，因此如果您需要更新窗体的任何控件或访问共享对象，必须实现自己的同步方法。

TsgcWebSocketServer | 服务器从客户端读取标头

当客户端连接到 WebSocket 服务器时，会发送一系列包含客户端连接信息的**请求头**。若要读取这些客户端请求头，可使用服务器组件的 **OnHandshake** 事件，该事件在服务器收到来自客户端的请求头且尚未发送响应之前调用。客户端头存储在 **TsgcWSConnectionServer** 的 **HeadersRequest** 属性中。

```
void OnServerHandshake(TsgcWSConnection *Connection, TStringList *Headers)
{
    ShowMessage(dynamic_cast<TsgcWSConnectionServer*>(Connection)->HeadersRequest->Text);
}
```

TsgcWebSocketHTTPServer

AResponseInfo.

简介

TsgcWebSocketHTTPServer 实现了服务器 WebSocket 组件，可以像 [TsgcWebSocketServer](#) 一样处理多线程客户端连接，并允许使用内置 HTTP 服务器提供 HTML 页面，在同一端口上共享 WebSocket 连接和 HTTP 请求。

请按照以下步骤配置此组件：

1. 在表单中拖放一个 TsgcWebSocketHTTPServer 组件
2. 设置端口（默认为 80）。如果您在防火墙后面，可能需要配置防火墙。
3. 设置允许的规范，默认情况下允许所有规范。

RFC6455：是标准且推荐的 WebSocket 规范。

Hixie76：这是一个草案版本，建议仅在需要支持 Safari 4.2 等旧版浏览器时才建立 Hixie76 连接。

```
AResponseInfo.ContentText := '<HTML><HEADER>TEST</HEAD><BODY>Hello!</BODY></HTML>';
```

OnHTTPUploadBeforeCreatePostStream: 该事件在读取请求头之后、创建 POST 流之前触发。

* 在某些情况下，由于未经请求的连接，您可能会遇到 CPU 占用率过高的问题。在这种情况下，对于 HTTP 请求返回 500 错误，对于未知协议请求关闭连接即可。

5. 创建一个过程并将属性 Active 设置为 True。

参考

- 属性 — 该组件提供 33 个属性。
- 方法 — 9 个公共方法。
- 事件 — 36 个可处理的事件。

指南

- HTTP
 - [HTTP 服务器请求](#)
 - [HTTP 分发文件](#)
 - [HTTP/2 服务器](#)
 - [HTTP/2 服务器推送](#)
 - [HTTP/2 备用服务](#)
 - [HTTP/2 服务器线程](#)
 - [HTTP Post 大文件](#)
 - [HTTP 404 无响应体错误](#)
- SSL
 - [Server SSL SChannel](#)
- 其他
 - [HTTP 服务器会话](#)

TsgcWebSocketHTTPServer | HTTP 服务器请求

使用 OnCommandGet 处理 HTTP 客户端请求。使用以下参数：

- **RequestInfo**：包含 HTTP 请求信息。
- **ResponseInfo**：HTTP 请求的 HTTP 响应。
 - **ContentText**：文本格式的响应。
 - **ContentType**：Content-Type 的类型。
 - **ResponseNo**：HTTP 响应号，例如：200。

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

OnBeforeCommand

使用此事件自定义 HTTP 响应。例如，如果您希望某些端点使用授权方案，而其他端点可以在不授权的情况下访问，请使用 options 参数来允许或禁用它。以下示例中，启用了 Authorization Basic，但当用户请求端点 /public 时，不需要授权。

```
void __fastcall TForm1::OnBeforeCommand(TsgcWSConnection* aConnection, TIdHTTPRequestInfo* ARequestInfo, TIdHTTPF
{
    if (ARequestInfo->Document == "/public")
        aOptions << hcoAuthorizedBasic;
}
```

TsgcWebSocketHTTPServer | HTTP 分发文件

当客户端请求文件时，会触发 **OnCommandGet** 事件，但您也可以使用 **DocumentRoot** 属性自动分发文件。

示例：如果您将 **DocumentRoot** 设置为 **c:/www/files**，每次请求新文件时，服务器将在此文件夹中搜索。如果文件存在，将自动分发。

TsgcWebSocketHTTPServer | HTTP/2 服务器

sgcWebSockets HTTP Server 允许您处理 HTTP/1.1 和 HTTP/2.0 请求，可通过服务器的 HTTP2Options 启用 HTTP/2 协议。

将 **HTTP2Options.Enabled = true** 设置为允许服务器接受 HTTP/2 协议请求。请求可以像处理 HTTP/1.1 协议一样被用户处理，[了解更多](#)。

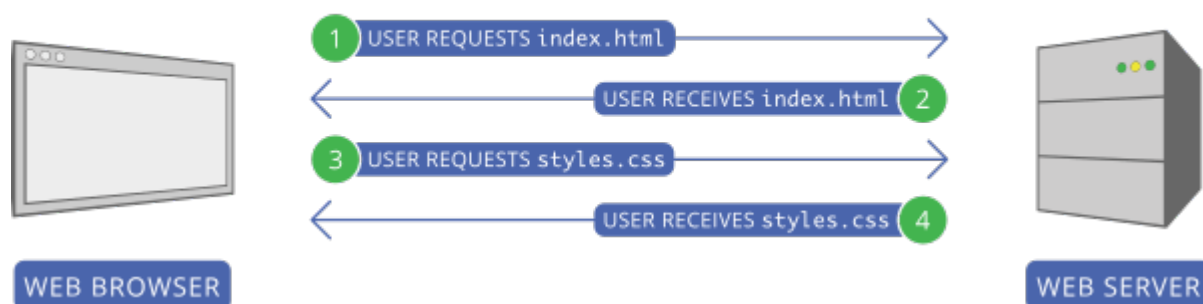
启用 HTTP/2 协议后，服务器仍将支持 HTTP/1.1 请求。

默认情况下，新建 HTTP/2 连接时不会触发 OnConnect 和 OnDisconnect 事件，但可以通过访问 HTTP2Options.Events 属性进行修改，在那里您可以自定义是否希望在每次新建 HTTP/2 连接和/或断开连接时收到通知。

TsgcWebSocketHTTPServer | HTTP/2 服务器推送

HTTP 通常采用请求/响应模式，客户端向服务器发送资源请求，服务器返回所请求资源或错误响应。通常，客户端（如浏览器）会向服务器发出一系列资源请求，由服务器提供这些资源。

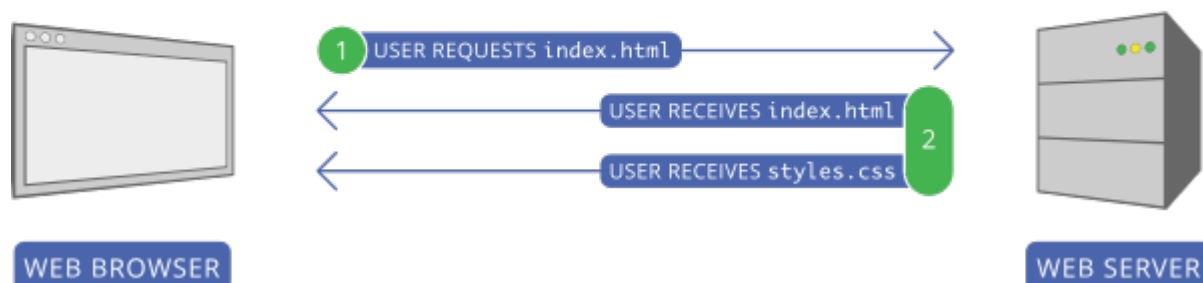
TYPICAL WEB SERVER COMMUNICATION



此方法的主要问题是客户端必须先发送请求以获取资源（例如 index.html），等待服务器发送响应，读取内容，然后才能发出所有其他请求（例如 styles.css）。

HTTP/2 服务器推送尝试解决此问题。当客户端请求某个文件时，若服务器判断该文件需要额外的文件，这些文件将被自动推送给客户端。

WEB SERVER COMMUNICATION WITH HTTP/2 SERVER PUSH



在上方截图中，客户端首先请求 index.html。服务器读取该请求后，将 2 个文件作为响应发送：index.html 和 styles.css，从而避免了再次请求 styles.css 的额外往返。

配置服务器推送

根据上方截图，您可以配置服务器，使其每次收到 /index.html 文件的请求时，都将 index.html 和 styles.css 一并发送。

使用 **PushPromiseAddPreLoadLinks** 方法将每个请求与推送承诺列表关联。

```

TsgcWebSocketHTTPServer *server = new TsgcWebSocketHTTPServer(this);
TStringList *oLinks = new TStringList();
try
{
    oLinks->Add("/styles.css");
    server->PushPromiseAddPreLoadLinks("/index.html", oLinks);
}
__finally
  
```

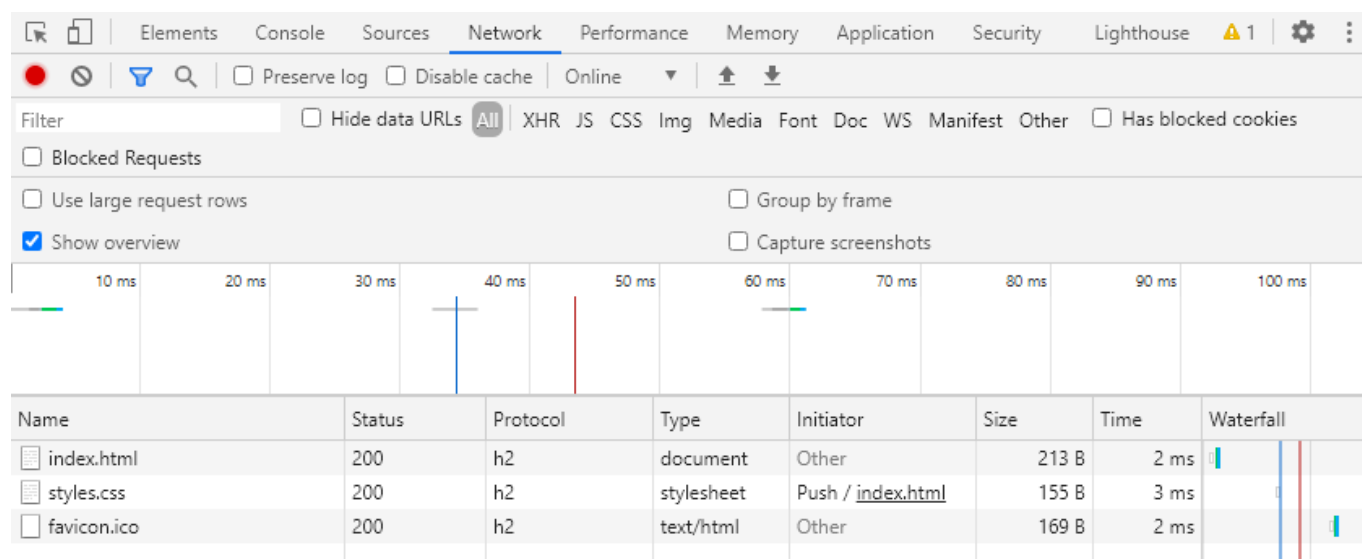
```

{
    oLinks->Free();
}

void OnCommandGet(TidContext *AContext, TidHttpRequestInfo *ARequestInfo, TidHttpResponseInfo *AResponseInfo)
{
    if (ARequestInfo->Document == "/index.html")
    {
        AResponseInfo->ContentText = "";
        AResponseInfo->ContentType = "text/html";
        AResponseInfo->ResponseNo = 200;
    }
    else if (ARequestInfo->Document == "/styles.css")
    {
        AResponseInfo->ContentText = "";
        AResponseInfo->ContentType = "text/css";
        AResponseInfo->ResponseNo = 200;
    }
}

```

使用 Chrome 开发者工具，您可以查看 styles.css 文件如何被推送到客户端。



Name	Status	Protocol	Type	Initiator	Size	Time	Waterfall
index.html	200	h2	document	Other	213 B	2 ms	
styles.css	200	h2	stylesheet	Push / index.html	155 B	3 ms	
favicon.ico	200	h2	text/html	Other	169 B	2 ms	

TsgcWebSocketHTTPServer | HTTP/2 备用服务

Alt-Svc HTTP 响应头用于通知客户端，同一资源可通过其他服务或协议访问，例如可用于告知 HTTP 客户端您的服务器支持 HTTP/2。

示例：如果您的服务器运行在本地 IP 127.0.0.1，并在 2 个端口上监听：80（非加密）和 443（加密）。您可以使用以下 HTTP 头通知客户端端口 443 支持 HTTP/2

```
Alt-Svc: h2=":443"
```

当 HTTP/2 已启用时，如果连接未运行在 HTTP/2 协议上，此标头将自动添加。
您可以使用属性 **HTTP2Options.AltSvc** 启用或禁用此功能。

TsgcWebSocketHTTPServer | HTTP/2 服务器线程

请参阅以下 HTTP 1.1 和 HTTP 2.0 之间的差异：

HTTP 1.1

在传统 HTTP 行为中，通过同一连接发出多个请求时，客户端必须等待每个请求的响应后才能发送下一个请求。这种顺序方式会显著增加网站资源的加载时间。为解决此问题，HTTP/1.1 引入了流水线功能，允许客户端在不等待服务器响应的情况下连续发送多个请求。服务器则按照接收请求的顺序依次响应客户端。

尽管流水线看似是一种解决方案，但它面临以下挑战：

- **服务器忽略或响应损坏**：一些服务器要么忽略了流水线请求，要么损坏了响应，导致通信不可靠。
- **队头阻塞**：管道中的第一个请求可能阻塞后续请求，导致其他请求处理延迟。这种现象称为队头阻塞，会导致页面加载速度变慢。

为了优化支持 HTTP/1.1 的服务器的页面加载速度，Web 浏览器实现了一种变通方案：向服务器并行打开六到八个连接，从而同时发送多个请求。这种并行化旨在缓解流水线相关问题并提升整体页面加载速度。

Web 浏览器选择六到八条并行连接是基于优化方面的考量。选择该数字的具体原因可能涉及资源利用率、网络效率与避免潜在瓶颈之间的权衡。

HTTP 2.0

为解决管道化（pipelining）遇到的限制，HTTP/2 引入了多路复用（multiplexing）功能。**多路复用**通过支持**更高效的通信**，实现了客户端与服务器之间通过**并发传输多个请求和响应**在**单一连接上的交互**。

HTTP/2 使用二进制分帧机制，这意味着 HTTP 消息被分解为称为帧的更小独立单元。这些帧可以在连接上交错发送，彼此独立。在接收端，帧被重新组装以重建原始 HTTP 消息。

这种二进制分帧机制是实现 HTTP/2 多路复用的基础。它使浏览器能够在同一连接上发送多个请求，而不会遇到阻塞问题。因此，Chrome 等浏览器对 HTTP/2 请求使用相同的连接 ID，从而在客户端和服务器之间实现高效、不间断的通信。

本质上，HTTP/2 的多路复用功能由二进制分帧机制实现，通过在单个连接上并发传输多个请求和响应，提升了客户端与服务器之间的数据交换效率和速度。

TsgcWebSocketHTTPServer

为了提高 HTTP/2 协议的性能，每次服务器收到新的 HTTP/2 请求时，默认在线程池（默认 32 个线程）中调度请求。这避免了单个连接发送多个并发请求时所需的顺序处理（在连接线程上下文中）造成的等待。

线程池的行为可通过以下属性进行配置。

- **HTTP2Options.PoolOfThreads.Enabled**：（默认为 false）启用此选项以在线程池中而非连接线程中分发 HTTP/2 请求。
- **HTTP2Options.Threads**：（默认 32）用于处理 HTTP/2 请求的线程数。请根据服务器的处理器数量设置此值。

要**精细调整请求**，选择哪些请求必须在线程池中处理（因为它们耗时较长），哪些可以在连接线程中处理，您可以使用事件 **OnHttp2BeforeAsyncRequest**。此事件在将请求排入线程池之前触发。使用参数 **Async** 设置请求是否为线程化。

```
void __fastcall TForm1::OnHTTP2BeforeAsyncRequest(TObject *Sender, TsgcWSConnection *Connection, const TIdHTTPRec
{
    if (ARequestInfo.Document == "/fast-request")
        Async = false;
}
```

TsgcWebSocketHTTPServer | 无响应正文的 404 错误

默认情况下，如果没有分配 `ContentText` 或 `ContentStream`，Indy 库会为 HTTP 响应添加内容体。如果您希望返回空响应体（例如 404 错误），可以使用以下方法。

创建一个不含内容的新 `TStringStream`，并将其赋值给 HTTP 响应的 `ContentStream` 属性。这样，HTTP 响应将在不包含默认 HTML 标签的情况下发送。

示例

```
private void OnCommandGet(TIdContext *AContext, TIdHTTPRequestInfo *ARequestInfo,
    TIdHTTPResponseInfo *AResponseInfo)
{
    AResponseInfo->ContentStream = new TStringStream("");
    AResponseInfo->ContentType = "text/html";
    AResponseInfo->ResponseNo = 404;
}
```

TsgcWebSocketHTTPServer | 会话

HTTP 是一种无状态协议，客户端请求文件，服务器发送响应，连接随即关闭。会话允许您存储关于客户端的信息，例如可在客户端登录期间使用。会话可以在一段时间不活动后销毁，也可以在客户端注销后手动销毁。

配置

TsgcWebSocketHTTPServer 中有一些属性用于在服务器组件中启用/禁用会话。最重要的属性有：

属性	描述
SessionState	这是必须启用的第一个属性，以便使用会话。如果不启用此属性，会话将无法工作
SessionTimeout	此处必须设置大于零的值（以毫秒为单位），作为会话保持活跃的最长时间。
AutoStartSession	会话可以自动创建（AutoStartSession = true）或手动创建（AutoStartSession = false）。如建会话，服务器将使用 RemoteIP 作为唯一标识符检查是否有活跃的存储会话。

```
TsgcWebSocketHTTPServer1->SessionState = true;
TsgcWebSocketHTTPServer1->SessionTimeout = 600000;
AutoStartSession = false;
```

创建会话

要创建新会话，必须创建一个**唯一**的新**会话 ID**。您可以使用任何值。**示例**：如果客户端正在验证身份，您可以使用 user + password + remoteip 作为会话 ID。然后，搜索会话列表以检查其是否已存在。如果不存在，则创建一个新会话。

创建新会话时，将调用 **OnSessionStart** 事件；会话关闭时，将触发 **OnSessionEnd** 事件。

```
void OnCommandGet(TidContext *AContext, TidHttpRequestInfo *ARequestInfo,
TidHttpResponseInfo *AResponseInfo)
{
    if (ARequestInfo->Document == "/")
    {
        AResponseInfo->ServeFile(AContext, "yourpathhere\\index.html");
    }
    else
    {
        // check if user is valid
        if (((ARequestInfo->AuthUsername == "user") && (ARequestInfo->AuthPassword == "pass")) == false)
        {
            AResponseInfo->AuthRealm = "Authenticate";
        }
        else
        {
            // create a new session id with authentication data
            string vID = ARequestInfo->AuthUsername + "_" + ARequestInfo->AuthPassword + "_" + ARequestInfo->RemoteIP;

            // search session
            TidHTTPSession oSession = TsgcWebSocketHTTPServer1->SessionList->GetSession(vID, ARequestInfo->RemoteIP);

            // create new session if not exists
            if (oSession == NULL)
            {
                oSession = TsgcWebSocketHTTPServer1->SessionList->CreateSession(ARequestInfo->RemoteIP, vID);
            }

            AResponseInfo->ContentText = "<html><head></head><body>Authenticated</body></html>";
            AResponseInfo->ResponseNo = 200;
        }
    }
}
```

```
}  
  }  
}
```

TsgcWebSocketHTTPServer | 流式视频

如果您想使用服务器对视频文件进行流式传输，可以使用 `IndyStreamFileVideo` 函数通过分块传输编码来流式传输该文件。

该函数接受以下参数：

- **AContext**：取自 `OnCommandGet` 事件。
- **AResponseInfo**：取自 `OnCommandGet` 事件。
- **aFileName**：要进行流式传输的视频的完整文件名。
- **ContentType**：默认为 `"video/mpeg"`。
- **aBufferSize**：默认值为 1024 字节。

示例：当用户访问文档 `/video.mp4` 时流式传输视频，该视频位于文件夹 `c:\videos\video.mp4` 中

```
void OnServerCommandGet(TidContext *AContext, TidHTTPRequestInfo *ARequestInfo, TidHTTPResponseInfo *AResponseInfo)
{
    if (ARequestInfo->Document == "/video.mp4") {
        IndyStreamFileVideo(AContext, AResponseInfo, "C:\\videos\\video.mp4");
    } else {
        AResponseInfo->ResponseNo = 404;
    }
}
```

服务器 SSL | Indy 服务器的 SChannel

基于 Indy 的服务器组件 (**TsgcWebSocketServer**、**TsgcWebSocketHTTPServer**) 可以使用 **Windows SChannel** (安全通道) 作为 TLS 提供程序, 而非 OpenSSL。SChannel 是 Windows 原生的 TLS 实现, 无需任何外部 DLL。

工作原理

当服务器组件启用了 SSL 且 IOHandler 设置为 **iohSChannel** 时, 服务器会创建一个 **TsgcdServerIOHandlerSSLSChannel** 实例, 使用 Windows SChannel API 处理所有 TLS 操作。对于每一个传入的客户端连接, 服务器通过 SChannel 提供程序执行 TLS 握手, 协商协议版本、密码套件, 并绑定已配置的证书。SChannel 从 **Windows 证书存储** 或 **PFX 文件** (.pfx / .p12) 读取证书。无需 PEM 文件, 也无需部署 OpenSSL 库。

配置

要在 Indy 服务器上启用 SChannel, 请配置以下属性:

1. 将 **SSL** 属性设置为 **True**。
2. 将 **SSLOptions.IOHandler** 设置为 **iohSChannel**。
3. 将 **SSLOptions.Version** 设置为所需的 TLS 版本 (tls1_2、tls1_3 等)。
4. 将 **SSLOptions.Port** 设置为用于安全连接的端口。
5. 使用下面描述的两种方法之一配置证书。

SChannel_Options 属性

SSLOptions.SChannel_Options 子属性公开了 SChannel 专用设置:

属性	描述
CertHash	安装在 Windows 证书存储中的证书指纹 (十六进制哈希值)。
CertStoreName	要搜索的证书存储: scsnMY (个人)、scsnRoot、scsnTrust、scsnCA。
CertStorePath	存储位置: scspStoreLocalMachine 或 scspStoreCurrentUser。
CipherList	可选的冒号分隔加密算法列表 (例如 CALG_AES_256:CALG_AES_128)。留空则使用系统默认值。
UseLegacyCredentials	当为 True 时, 使用旧版 SCHANNEL_CRED 结构代替 SCH_CREDENTIALS。对于不支持更新 API 的旧版 Windows, 请启用此选项。

此外, 通用 **SSLOptions** 属性 **CertFile** 和 **Password** 在从 PFX 文件加载证书时使用。

来自 Windows 存储的证书

如果证书已安装在 Windows 证书存储中, 请提供证书 **指纹** 并指明其所在位置。
要查找指纹, 请打开 **PowerShell** 并运行:

```
dir cert:\localmachine\my
```

Thumbprint 列显示您所需的十六进制哈希值。

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint Subject
-----
C12A8FC8AE668F866B48F23E753C93D357E9BE10 CN=*.mydomain.com
```



```
oServer = new TsgcWebsocketHTTPServer();
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->Version = tls1_2;
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->SSLOptions->SChannel_Options->CertHash = "C12A8FC8AE668F866B48F23E753C93D357E9BE10";
oServer->SSLOptions->SChannel_Options->CertStoreName = scsnMY;
oServer->SSLOptions->SChannel_Options->CertStorePath = scspStoreLocalMachine;
oServer->Active = true;
```

来自 PFX 文件的证书

如果您有 PFX (.pfx 或 .p12) 证书文件，请在 SSLOptions 上设置 **CertFile** 和 **Password** 属性。SChannel 将在启动时导入证书。

```
oServer = new TsgcWebsocketHTTPServer();
oServer->SSL = true;
oServer->SSLOptions->IOHandler = iohSChannel;
oServer->SSLOptions->Version = tls1_2;
oServer->SSLOptions->Port = 443;
oServer->Port = 443;
oServer->SSLOptions->CertFile = "c:\\certificates\\server.pfx";
oServer->SSLOptions->Password = "mypassword";
oServer->Active = true;
```

如果您有 PEM 证书和私钥，请先使用 OpenSSL 将其转换为 PFX 格式：

```
openssl pkcs12 -inkey server.key -in server.crt -export -out server.pfx
```

TLS 版本

使用 **SSLOptions.Version** 属性控制服务器接受的 TLS 版本：

值	描述
tls1_0	TLS 1.0（不推荐）
tls1_1	TLS 1.1
tls1_2	TLS 1.2（推荐）
tls1_3	TLS 1.3
tlsUndefined	接受 TLS 1.0、1.1 和 1.2

密码套件列表

默认情况下，SChannel 使用系统密码配置。您可以通过将 **SChannel_Options.CipherList** 设置为以冒号分隔的算法名称列表来限制允许的密码，例如：

```
CALG_AES_256:CALG_AES_128
```

将此属性留空以使用 Windows 默认值。

旧版凭据

Windows Server 2019 及更早版本可能不支持较新的 **SCH_CREDENTIALS** API。如果服务器无法在较旧的 Windows 版本上启动，请将 **SChannel_Options.UseLegacyCredentials** 设置为 **True** 以改用旧版 **SCHANNEL_CRED** 结构。

该组件在大多数情况下会自动检测 Windows 版本，但如有需要，您也可以强制启用传统模式。

相对于 OpenSSL 的优势

- **无需外部 DLL**：SChannel 已内置于 Windows，因此无需部署 libeay32.dll / ssleay32.dll 或 libcrypto / libssl。
- **Windows 证书存储集成**：使用操作系统已安装和管理的证书。
- **自动更新**：TLS 改进和安全补丁通过 Windows Update 应用。

注意事项

- SChannel 仅在 **Windows** 上可用。对于跨平台服务器，请使用 OpenSSL（iohOpenSSL）。
- 服务器必须拥有与证书关联的**私钥**。使用 Windows 存储方法时，证书必须连同私钥一起导入。
- 对于使用证书存储方式的生产服务器，推荐使用 **本地计算机** 存储 (scspStoreLocalMachine)，以确保无论服务以哪个用户账户运行，证书均可访问。
- 仅支持 **PFX** (.pfx/.p12) 证书文件。如果您有 PEM 文件，请先将其转换为 PFX 格式。

TsgcWebSocketServer_HTTPAPI

HTTP 服务器 API 使应用程序能够在不使用 Microsoft Internet Information Server (IIS) 的情况下通过 HTTP 进行通信。

简介

HTTP 服务器 API 使应用程序无需使用 Microsoft Internet Information Server (IIS) 即可通过 HTTP 进行通信。应用程序可注册以接收特定 URL 的 HTTP 请求、接收 WebSocket 请求并发送 WebSocket 响应。HTTP 服务器 API 包含 SSL 支持，使应用程序无需 IIS 即可通过安全 HTTP 连接交换数据，并且还设计为与 I/O 完成端口协同工作。

服务器支持以下协议：

- WebSocket（需要 Windows 8 或更高版本）
- HTTP 1.1
- HTTP/2（需要 Windows 2016+ 或 Windows 10+）。

默认情况下，此组件要求您的应用程序以管理员模式运行以进行 URL 注册。如果 URL 已通过 netsh 等外部工具注册，则可以不以管理员权限运行，禁用 `BindingOptions.ConfigureSSLCertificate` 属性即可无需管理员权限启动应用程序。

将 `FastMM4/FastMM5` 设置为项目的第一个单元。

请按照以下步骤配置此组件：

1. 在表单中放置一个 `TsgcWebSocketServer_HTTPAPI` 组件
2. 定义监听地址和端口：

```
Server->Host = "127.0.0.1";  
Server->Port = 80;
```

3. 设置允许的规范，默认情况下允许所有规范。

RFC6455：是标准且推荐的 WebSocket 规范。

Hixie76：这是一个草案版本，建议仅在需要支持 Safari 4.2 等旧版浏览器时才建立 Hixie76 连接。

5. 创建一个过程并将属性 `Active` 设置为 `True`

URL 预留

HTTP.SYS 服务器使用 URL 预留来分配 HTTP.SYS 服务器将使用的 URL 端点。

基本 URL 预留

这是配置服务器最简单的模式，基本上只需设置 HTTP.SYS 服务器处理的 Host 和 Port 即可。

示例：如果您的服务器运行在 IP 127.0.0.1 和端口 80，只需设置以下属性

```
Server->Host = "127.0.0.1";  
Server->Port = 80;
```

如果服务器运行于多个 IP 上，且您希望绑定到多个 IP，请使用 **NewBinding** 方法。首先清除 Host 和 Bindings 属性，然后使用 `NewBinding` 方法定义所有服务器绑定。

```
Server->Host = "";  
Server->Bindings->Clear;  
Server->Bindings->NewBinding("127.0.0.1", 80, "");  
Server->Bindings->NewBinding("80.50.55.11", 80, "");
```

如果服务器要求 SSL 连接，请执行以下操作以定义用于处理 SSL 连接的 Host 和 Port。

```
Server->Host = "127.0.0.1";  
Server->Port = 443;  
Server->SSL = true;  
Server->SSLOptions->Hash = "CERTIFICATE_HASH";
```

如果服务器需要具有多个 IP 地址的 SSL 连接，请先清除 Host 和 Bindings 属性，然后注册新的 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");
```

参考

- 属性 — 组件上提供 27 个属性。
- 方法 — 9 个公共方法。
- 事件 — 可处理的 23 个事件。

指南

- 配置
 - [URL 保留](#)
- 连接
 - [OnDisconnect 未触发](#)
- SSL
 - [HTTPAPI 服务器 SSL](#)
 - [自签名证书](#)
- HTTP
 - [自定义头](#)
 - [发送文本响应](#)
 - [发送文件响应](#)
 - [上传大文件](#)
- HTTP/2
 - [禁用 HTTP/2](#)

HTTPAPI | URL 预留

HTTP.SYS URL 预留是 Windows 操作系统中的一项功能，允许用户为其应用程序或服务预留特定的统一资源定位符（URL）。使用 HTTP.SYS 预留 URL 后，操作系统将拦截该 URL 的所有传入 HTTP 请求，并将其路由到指定的应用程序或服务。

要使用 HTTP.SYS 保留 URL，应用程序或服务必须首先通过调用 HTTP API 向 HTTP.SYS 驱动程序注册该 URL。应用程序或服务需指定 URL、HTTP 方法（如 GET、POST）及认证要求等附加设置。

URL 注册后，HTTP.SYS 将拦截该 URL 的所有传入 HTTP 请求，并根据 URL 和方法查找已注册的应用程序或服务。如果找到匹配的应用程序或服务，HTTP.SYS 驱动程序将把请求传递给该应用程序或服务进行处理。

NETSH 命令

注册 URL

在此示例中，正在为用户 DOMAIN\user 注册 URL http://example.com:80/。您可以将其替换为所需的 URL 和用户。

```
netsh http add urlacl url=http://example.com:80/ user=DOMAIN\user
```

删除 URL

在此示例中，URL http://example.com:80/ 正在被删除。您可以将其替换为要删除的 URL。

```
netsh http delete urlacl url=http://example.com:80/
```

显示所有 URL

此命令将显示系统上所有已注册 URL 保留项的列表。

```
netsh http show urlacl
```

TsgcWebSocketServer_HTTPAPI

HTTP.SYS 服务器在启动时自动注册 URL，使用以下参数和方法完成注册。

- **Host 和 Port**：若 Host 不为空且 Port 不为零，服务器将尝试注册该 URL。示例：URL https://127.0.0.1:5000 将使用以下属性注册
 - Host = '127.0.0.1';
 - Port = 5000
 - SSL = True
- **NewBinding**：使用此方法注册一个或多个 URL。
 - 注册 URL https://127.0.0.1:5000 --> NewBinding('127.0.0.1', 5000, '/', True)
 - 注册 URL http://+:5000/ws/ --> NewBinding('+', 5000, '/ws/')

URL 注册在以下情况需要管理员权限：

- 端口号低于 1024
- 主机是通配符 "+", 而非 IP 地址。

如果您想为服务器的所有 IP 地址注册 443 端口，只监听端点 "/ws/"，但不希望以管理员权限运行服务器，请执行以下步骤：

- 使用 netsh 注册 URL
 - netsh http add urlacl url=https://+:443/ws/ user=DOMAIN\user
- 使用以下绑定配置服务器
 - NewBinding('+', 443, '/ws/', True);
- 禁用 ConfigureSSLCertificate 属性
 - TsgcWebSocketServer_HTTPAPI.BindingOptions.ConfigureSSLCertificate = false;
- 配置 SSL 证书
 - [HTTPAPI 服务器 SSL](#)

HTTPAPI | FineTune

TsgcWSServer_HTTPAPI 公开了一个名为 `FineTune` 的已发布属性，类型为 `TsgcServerHTTPAPI_FineTune`。它汇集了影响 Windows HTTP Server API (`http.sys`) 如何对请求进行排队、分发和完成的底层内核模式调节项。

QueueLength

包装 `HttpServerQueueLengthProperty` 内核设置。它控制 `http.sys` 在服务器从队列中取出请求之前在内核模式队列中保存的最大待处理请求数。当队列满时，内核直接以 **503 Service Unavailable** 响应新连接，而不会进入用户模式。

Type：ULONG。

默认值：1000（Windows 内核默认值）。

范围：最多 65535 个请求。

改进效果：提高此值可防止内核在突发工作负载下拒绝合法流量，例如 IoT 设备群在网络中断后的重连、市场数据广播客户端在提供商重置后的重连，或所有工作节点在共享 cron 任务触发时同时连接的场景。

SkipIOCPOnSuccess

当设置为 `True` 时，服务器使用标志 `FILE_SKIP_COMPLETION_PORT_ON_SUCCESS` 和 `FILE_SKIP_SET_EVENT_ON_HANDLE` 在请求队列句柄上调用 `SetFileCompletionNotificationModes`。同步完成的请求队列上的重叠 I/O 操作将不再向 I/O 完成端口发布额外的完成数据包。

类型：Boolean。

默认值：False（未设置任何标志，保留当前行为）。

改进内容：当调用同步返回 `NO_ERROR` 时，消除热请求路径上的内核到用户模式切换——工作线程在调用线程上内联分发完成，而无需等待 IOCP 数据包。这是 Microsoft 参考 HTTP Server High Performance 示例所使用的模式。该标志旨在与 `OperatingMode = ompHighPerf` 配合使用。

API 在运行时通过 `GetProcAddress` 从 `kernel32` 解析。在 Windows XP 及更早版本（Delphi 7 运行时目标）上，该入口点不存在，包装器返回 `False`，服务器在未设置该标志的情况下运行。

OperatingMode

选择两种接受/调度架构之一。

Type：TsgcHTTPAPIOperatingMode = (ompClassic, ompHighPerf)。

默认值：ompClassic。

- **ompClassic** — 单个接受线程同步调用 `HttpReceiveHttpRequest`，并通过 `PostQueuedCompletionStatus` 将每个请求手动分发到 IOCP 工作线程池。这是历史上的默认行为。
- **ompHighPerf** — 实现 Microsoft 参考 HTTP 服务器高性能模式。acceptor 线程被移除。启动时，服务器在绑定 IOCP 的队列上预发布 `ThreadPoolSize x HighPerfAcceptsPerWorker` 个异步接收（默认为 $32 \times 4 = 128$ 个并发挂起接收）。每个工作线程内联处理其完成通知并重新发布一个异步接收，维持一个滑动窗口直到关闭。

改进说明：当服务器遇到深度单流管道（大帧上传/下载）或大量并发客户端（数百至数千）时，`ompHighPerf` 的优势最为明显。预投递接收窗口可吸收突发流量而无需按连接分配，内联分发消除了接受者切换瓶颈。对于低流量 API 和开发环境，请保留默认的 `ompClassic`，在轻负载下维护 128 个预投递上下文的开销大于其带来的收益。此模式只能在服务器激活前更改。

HighPerfAcceptsPerWorker

控制当 `operatingMode = ompHighPerf` 时每个 IOCP 工作线程预发布的异步接收数量。在 `ompClassic` 模式下忽略此值。服务器维护的并发未完成接收总数等于 `ThreadPoolSize x HighPerfAcceptsPerWorker`。

Type：整数。

默认值：4。

改进内容：更深的每工作线程窗口允许服务器在热路径上无需分配新上下文即可吸收更大的突发入站请求。对于高并发部署（IoT 设备群、市场数据分发、扇出代理），请适当提高该值；代价是内存——每个预投递接收持有有一个约 16 KB 的请求缓冲区，直到完成前一直保留。默认值 4 是经过 MSDN HP 示例验证的保守折中值。

示例

以下代码片段为高并发 IoT 后端配置了 HTTP.sys 服务器：使用大型内核队列以吸收重连风暴，使用 HighPerf 分发模式并扩展预投递接收窗口，并启用内联完成分发。

```
#include "sgcWebSocket_Server_HTTPAPI.hpp"
#include "sgcWebSocket_HTTPAPI_Server.hpp" // TsgcHTTPAPIOperatingMode
TsgcWSServer_HTTPAPI *oServer = new TsgcWSServer_HTTPAPI(NULL);
oServer->Host = "0.0.0.0";
oServer->Port = 8080;
// absorb 10,000-device reconnect bursts before kernel-level 503
oServer->FineTune->QueueLength = 10000;
// switch from single-acceptor to pre-posted IOCP workers
oServer->FineTune->OperatingMode = ompHighPerf;
// widen the per-worker pre-posted receive window (32 threads * 8 = 256)
oServer->FineTune->HighPerfAcceptsPerWorker = 8;
// dispatch inline on sync-success completions; skip the IOCP round-trip
oServer->FineTune->SkipIOCPOnSuccess = true;
oServer->Active = true;
```

对于典型的内部或低流量 API，请将每个 FineTune 属性保留为默认值：

```
TsgcWSServer_HTTPAPI *oServer = new TsgcWSServer_HTTPAPI(NULL);
oServer->Host = "localhost";
oServer->Port = 8080;
// FineTune defaults: QueueLength=1000, SkipIOCPOnSuccess=False,
//                      OperatingMode=ompClassic, HighPerfAcceptsPerWorker=4
oServer->Active = true;
```


TsgcWebSocketServer_HTTPAPI | HTTPAPI Server SSL

服务器可配置为使用 **SSL 证书**。要获取带有服务器证书的生产服务器，您必须从**知名提供商**（Namecheap、GoDaddy、Thawte 等）**购买证书**。出于**测试目的**，您可以使用**自签名证书**（参见使用自签名证书的 Demos/Chat 示例）。请阅读以下文章：[如何创建自签名证书](#)。

获得证书后，您必须配置服务器以指定用于加密连接的证书。

证书哈希

首先，您需要知道证书的哈希值。在 **powershell** 中，只需对证书容器运行 **dir** 命令即可轻松获取证书的哈希值。

```
dir cert:\localmachine\my
```

哈希是十六进制的**Thumbprint**（指纹）值。

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint                               Subject
-----
C12A8FC8AE668F866B48F23E753C93D357E9BE10  CN=*.mydomain.com
```

获取 Thumbprint 值后，只需在 **TsgcWebSocketServer_HTTPAPI.TLSOptions.Hash** 属性中进行设置即可。

设置哈希后，只需将 **TsgcWebSocketServer_HTTPAPI.SSL = true**，您的服务器即可开始运行。

如果您想使用 netsh 手动注册证书，请使用以下命令：

```
netsh http add sslcert ipport=<IP>:<PORT> certhash=<THUMBPRINT> appid="{<GUID>}"
```

<IP>：指定绑定的本地 IP 地址。请勿使用通配符绑定，须使用有效的 IP 地址。

<PORT>：指定绑定的端口。

<THUMBPRINT>：X.509 证书指纹。

<GUID>：开发人员生成的 GUID，仅用于信息性地标识应用程序。

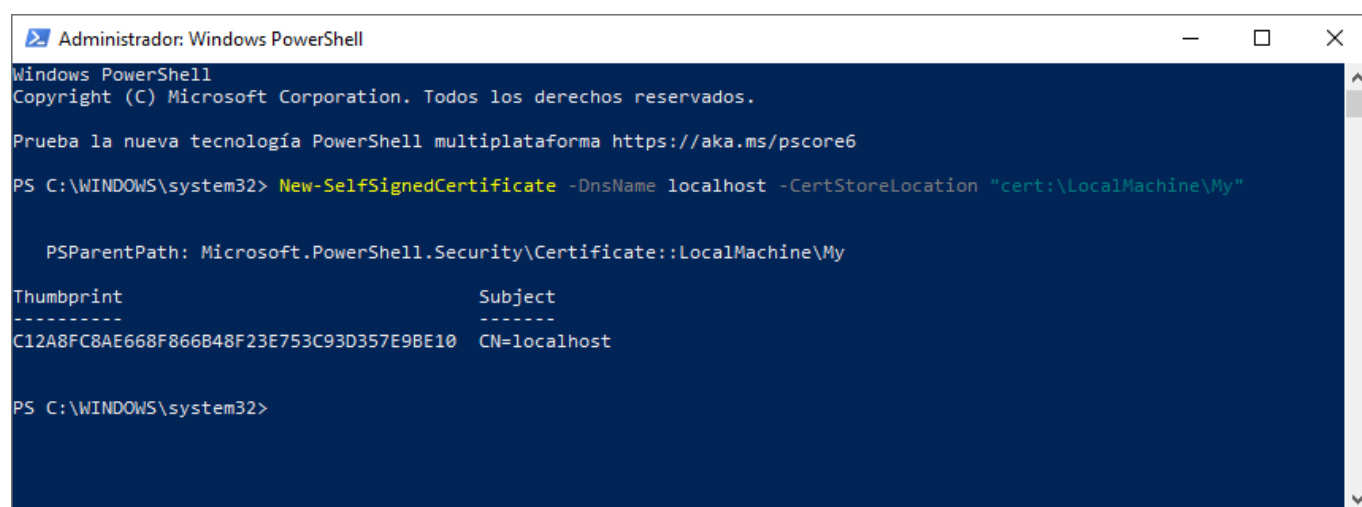
TsgcWebSocketServer_HTTPAPI | 自签名证书

如果您需要用于自己测试的证书，可以在测试机上创建自签名证书。请按照以下步骤操作：

1. 以**管理员**身份运行 **Powershell**。
2. 运行以下命令创建证书：

```
New-SelfSignedCertificate -DnsName localhost -CertStoreLocation "cert:\LocalMachine\My"
```

如果成功，您将收到关于新建证书的确认信息。只需复制指纹并粘贴到 **TsgcWebSocketServer_HTTPAPI.TLSOptions.Hash** 属性中。



```
Administrador: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\WINDOWS\system32> New-SelfSignedCertificate -DnsName localhost -CertStoreLocation "cert:\LocalMachine\My"

    PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\My

Thumbprint                               Subject
-----
C12A8FC8AE668F866B48F23E753C93D357E9BE10  CN=localhost

PS C:\WINDOWS\system32>
```

3. 可选，您可以将自签名证书添加为**受信任的证书颁发机构**

```
Run MMC -32 as administrator
```

- 3.1. 选择 **文件 / 添加或删除管理单元**
- 3.2. 选择证书，然后单击添加
- 3.3. 选择计算机账户并点击下一步。
- 3.4. 选择“本地计算机”并点击确定。您将看到您的证书。
- 4.5. 从**个人/证书**中选择您的证书，并粘贴到 **受信任的根证书颁发机构/证书**。

TsgcWebSocketServer_HTTPAPI | 禁用 HTTP/2

HTTP/2 协议在 **Server 2016+** 和 **Windows 10+** 操作系统上默认启用。在一些较旧的浏览器或 HTTP 客户端中，您可能会遇到错误，因为 该协议不完全受支持。您可以通过禁用 HTTP/2 协议来避免这些错误。

如何禁用 HTTP/2

- 打开 Windows 注册表编辑器
- 转到以下注册表项：

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\HTTP\Parameters

- 添加以下 DWORD 值 并将两个值均设为零。
 - EnableHttp2Tls
 - EnableHttp2Cleartext
- 重启计算机。

TsgcWebSocketServer_HTTPAPI | 自定义头

您可以使用响应对象的 **CustomHeaders** 属性自定义 HTTP.SYS 服务器的响应。

使用换行符分隔的头部名称和头部值设置 CustomHeaders 的值。

示例：如果您想添加以下头部，请参见以下示例代码

```
Access-Control-Allow-Origin: *  
Access-Control-Allow-Methods: GET, POST, OPTIONS, PUT, PATCH, DELETE
```

```
private void OnHttpRequest(TsgcWSConnection_HTTPAPI *aConnection, const THttpRequest *aRequestInfo,  
    ref THttpResponse *aResponseInfo)  
{  
    aResponseInfo->ResponseNo = 200;  
    aResponseInfo->CustomHeaders = "Access-Control-Allow-Origin: *\r\n" + "Access-Control-Allow-Methods: " +  
        "GET, POST, OPTIONS, PUT, PATCH, DELETE";  
}
```

TsgcWebSocketServer_HTTPAPI | 发送文本响应

使用 `OnHttpRequest` 事件处理 HTTP 请求。

类 `THttpRequest` 包含 HTTP 请求数据。

- **文档**：对等方尝试访问的文档。
- **Method**：HTTP 方法（'GET'、'POST' 等）
- **Headers**：HTTP 请求的头部。
- **AcceptEncoding**：接受编码变量，示例："gzip, deflate, br"。
- **ContentType**：示例："text/html"
- **Content**：请求的内容（若存在）。
- **QueryParams**：查询参数。
- **Cookies**：若存在，则为 Cookie。
- **ContentLength**：内容的大小。
- **AuthExists**, **AuthUsername**, **AuthPassword**：身份验证请求数据。
- **Stream**：如果 HTTP 请求有正文，这是正文的流。

`THttpResponse` 类包含 HTTP 响应数据。

- **ContentText**：以文本形式表示的响应内容。
- **ContentType**：示例："text/html"。若要将 ContentText 以 UTF8 编码，请设置 `charset='utf-8'`。示例：`text/html; charset=utf-8`
- **CustomHeaders**：如果需要发送自定义头部，请使用此变量
- **AuthRealm**：若服务器需要身份验证，请设置此变量。
- **ResponseNo**：HTTP 响应编号。示例：200 表示响应成功。
- **ContentStream**：如果响应包含流，请在此处设置（无需释放该流，它将被自动释放）。
- **FileName**：如果响应是文件名，请在此处设置文件的完整路径。
- **Date**、**Expires**、**LastModified**：响应的日期时间变量。
- **CacheControl**：允许自定义缓存行为。

示例：如果服务器收到文档"/test.html"的 GET 请求，则发送 OK 响应；否则，如果是对其他文档的 GET 请求，则发送 404，如果是不同的方法，则发送错误 500。

```
void __fastcall OnHttpRequest(TsgcWSConnection_HTTPAPI *aConnection,
    const THttpRequest *aRequestInfo,
    THttpResponse *aResponseInfo)
{
    if (aRequestInfo->Method == "GET")
    {
        if (aRequestInfo->Document == "/test.html")
        {
            aResponseInfo->ResponseNo = 200;
            aResponseInfo->ContentText = "OK";
            aResponseInfo->ContentType = "text/html; charset=UTF-8";
        }
        else
        {
            aResponseInfo->ResponseNo = 404;
        }
    }
    else
    {
        aResponseInfo->ResponseNo = 500;
    }
}
```

TsgcWebSocketServer_HTTPAPI | 发送文件响应

如果您想将文件作为 HTTP 请求的响应发送，请使用 **THttpRequestResponse** 对象的 **FileName** 属性。

```
void __fastcall OnHttpRequest(TsgcWSConnection_HTTPAPI *aConnection,
    const THttpRequest *aRequestInfo,
    THttpRequestResponse *aResponseInfo)
{
    if (aRequestInfo->Method == "GET")
    {
        if (aRequestInfo->Document == "/test.zip")
        {
            aResponseInfo->ResponseNo = 200;
            aResponseInfo->FileName = "c:\\download\\test.zip";
            aResponseInfo->ContentType = "application/zip";
        }
        else
        {
            aResponseInfo->ResponseNo = 404;
        }
    }
    else
    {
        aResponseInfo->ResponseNo = 500;
    }
}
```

可续传下载

当服务器响应资源的特定部分（范围）请求而非发送整个文件时，将使用 HTTP 206 部分内容响应。这通常用于可续传的下载、媒体流传输和大文件传输。

工作原理：

客户端请求部分资源：客户端（浏览器、下载器或媒体播放器）发送一个指定所需字节范围的 **Range** 头。示例请求：

```
GET /video.mp4 HTTP/1.1
Host: example.com
Range: bytes=1000-5000
```

此请求获取 video.mp4 的第 1000 至 5000 字节。

服务器响应 HTTP 206：如果服务器支持范围请求，它将以 206 Partial Content 响应并包含 **Content-Range** 标头。示例响应：

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 1000-5000/1000000
Content-Length: 4001
Content-Type: video/mp4
```

Content-Range 头部显示：

服务范围（1000-5000）
文件的总大小（1000000 字节）。
Content-Length 头的大小为返回部分的大小（4001 字节）。

客户端可以请求更多块：

客户端可以为不同的部分发送多个请求。
这实现了可恢复下载和高效流传输。

```

void __fastcall OnHttpRequest(TidContext* AContext, TidHttpRequestInfo* ARequestInfo, TidHttpResponseInfo* AResponseInfo)
{
    std::unique_ptr<TFileStream> oStream(new TFileStream("test.pdf", fmOpenRead | fmShareDenyWrite));
    std::unique_ptr<TidEntityRangeStrings> oRanges(new TidEntityRangeStrings(nullptr));
    try {
        oRanges->Text = ARequestInfo->RawHeaders->Values["Range"];
        AResponseInfo->ContentType = "application/pdf";
        if (oRanges->Count > 0) {
            AResponseInfo->ResponseNo = 206; // Partial Content
            AResponseInfo->AcceptRanges = "bytes";
            AResponseInfo->ContentRangeStart = oRanges->Items[0]->StartPos;
            AResponseInfo->ContentRangeEnd = oRanges->Items[0]->EndPos;
            AResponseInfo->ContentRangeInstanceLength = oStream->Size;

            AResponseInfo->ContentStream = new TidHTTPRangeStream(oStream.release(),
                AResponseInfo->ContentRangeStart, AResponseInfo->ContentRangeEnd);
        } else {
            AResponseInfo->ResponseNo = 200; // OK
            AResponseInfo->ContentStream = oStream.release();
        }
    } catch (...) {
        // Handle exceptions
    }
}

```

TsgcWebSocketServer_HTTPAPI | OnDisconnect 未触发

初次使用 HTTPAPI 服务器时，很常见的情况是：OnDisconnect 事件并不会在客户端关闭连接时立即触发。原因在于 HTTPAPI 服务器的工作方式与 Indy 等其他服务器略有不同。在 **Indy 服务器** 中，**每个连接都有一个专属线程**，该线程每隔 x 毫秒检查一次**连接是否活跃**。而 **HTTPAPI 服务器** 使用**线程池** 处理所有连接，**不会对每个连接逐一检查是否活跃**。

为了在客户端关闭连接时收到通知，请进行以下配置：

1. 如果您使用 [TsgcWebSocketClient](#)，请设置 **Options.CleanDisconnect := True**。这意味着在关闭连接之前，客户端将尝试向服务器发送连接即将关闭的通知。如果服务器收到此消息，将调用 OnDisconnect 事件。
2. 对于其他断开连接的情况，唯一的解决方案是向套接字写入数据；如果失败，则连接断开。在 HTTPAPI 服务器上**启用 HeartBeat**，例如将间隔设置为 60 秒，超时设置为 0。这种配置意味着每 60 秒对所有连接发送一次 ping，如果有连接已断开，将触发 **OnDisconnect** 事件。您可以设置较低的 HeartBeat.Interval 值，但不要设置得太低（例如 1 秒太低），因为这会影响服务器性能。

TsgcWebSocketClient_WinHTTP

TsgcWebSocketClient_WinHTTP 实现了一个可以连接到 WebSocket 服务器的客户端 VCL WebSocket 组件。

简介

TsgcWebSocketClient_WinHTTP 实现了一个客户端 VCL WebSocket 组件，可连接到 WebSocket 服务器。它基于 WinHTTP API，需要 Windows 8 或更高版本。请按照以下步骤配置此组件：

1. 将 TsgcWebSocketClient_WinHTTP 组件拖放到窗体上
2. 设置 Host 和 Port（默认为 80）以连接到可用的 WebSocket 服务器。您可以设置 URL 属性，Host、Port、Parameters 等将从 URL 自动更新。例如：wss://127.0.0.1:8080/ws/ 将得到：

```
oClient = new TsgcWebSocketClient_WinHTTP();  
oClient->Host = "127.0.0.1";  
oClient->Port = 80;  
oClient->TLS = true;  
oClient->Options->Parameters = "/ws/";
```

3. 您可以选择是否启用 TLS（安全连接），默认为禁用。
4. 以下事件可用于自定义 WebSocket 客户端流程：
5. 创建一个过程并将 Active 属性设置为 True。

参考

- 属性 — 该组件提供 16 个属性。
- 方法 — 8 个公共方法。
- 事件 — 可处理 9 个事件。

TsgcWebSocketFirewall

TsgcWebSocketFirewall 实现了一个全面的防火墙组件，保护 WebSocket 服务器免受各种安全威胁。

简介

TsgcWebSocketFirewall 实现了一个全面的防火墙组件，可保护 WebSocket 服务器免受各种安全威胁。它提供十六种保护模块，包括 IP 黑名单/白名单过滤、带自动 IP 封禁的暴力破解防护、SQL 注入检测、XSS（跨站脚本）检测、连接速率限制、消息泛洪防护、基于 GeoIP 国家的过滤、基于 IP 的机器人检测、动态威胁评分、路径遍历检测、命令注入检测、载荷大小限制、WebSocket 特定防护（Origin 验证、帧大小限制、子协议过滤）、渐进式封禁升级、自定义规则引擎以及实时统计。

防火墙通过 TsgcWebSocketServer、TsgcWebSocketHTTPServer 和 TsgcWebSocketServer HTTPAPI 上提供的 **Firewall** 属性与服务器组件自动集成。一经赋值并配置完毕，即可在无需手动连接事件的情况下拦截连接和消息。

快速入门

1. 在窗体上放置一个 **TsgcWebSocketFirewall** 组件。

2. 配置所需的保护模块：

黑名单：将 Enabled 设为 True，并在 IPs 列表中添加 IP 或 CIDR 范围（例如"10.0.0.0/8"、"192.168.1.100"）。

Whitelist：将 Enabled 设置为 True 并添加受信任的 IP。列入白名单的 IP 绕过所有其他检查。

BruteForce：将 Enabled 设为 True。配置 MaxAttempts（默认 5）、TimeWindowSec（默认 60）和 BanDurationSec（默认 300）。

SQLInjection：将 Enabled 设置为 True。检测消息中常见的 SQL 注入模式。

XSS：将 Enabled 设置为 True。检测消息中的跨站脚本模式。

RateLimit：将 Enabled 设为 True。限制每个 IP 的并发连接数（默认每 60 秒 10 个）。

FloodProtection：将 Enabled 设置为 True。限制每个 IP 每秒的消息数（默认 100）。

PayloadLimit：将 Enabled 设置为 True。阻止超过 MaxSizeBytes（默认 1MB）的消息。

PathTraversal：将 Enabled 设置为 True。检测目录遍历模式。

CommandInjection：将 Enabled 设为 True。检测 Shell 命令注入模式。

ThreatScore：将 Enabled 设置为 True。根据违规情况动态为 IP 分配分值。

BanEscalation：将 Enabled 设置为 True。对于屡次违规者，逐步升级封禁时长。

GeoIP：将 Enabled 设置为 True。按国家代码阻止或允许连接。

BotDetection：将 Enabled 设置为 True。将连接 IP 分类为经验证的爬虫、数据中心主机、疑似机器人、列入黑名单或人类用户。

WebSocketProtection：将 Enabled 设为 True。验证来源、帧大小和子协议。

CustomRules：将 Enabled 设为 True。用于复杂过滤的可配置规则引擎。

3. 将防火墙分配给服务器组件：

```
sgcWebSocketHTTPServer1.Firewall := sgcWebSocketFirewall1;
```

4. 启动服务器。防火墙自动保护所有连接和消息。

属性

属性	描述
已启用	启用或禁用防火墙。默认值：True。
黑名单	IP 黑名单配置。包含 Enabled（布尔值）和 IPs（TStringList）属性。支持精确 IP 和 CIDR 表示法（例如"192.168.0.0/16"）。
白名单	IP 白名单配置。白名单中的 IP 绕过所有安全检查。与 Blacklist 结构相同。
BruteForce	暴力破解保护。属性：Enabled、MaxAttempts（默认 5）、TimeWindowSec（默认 60）、BanDurationSec（默认 300）。自动封禁超过尝试次数限制的 IP。
SQLInjection	SQL 注入检测。属性：Enabled、Action（faDeny/faAllow/faLog）、CustomPatterns（用于附加模式的 TStringList）。
XSS	跨站脚本检测。属性：Enabled、Action（faDeny/faAllow/faLog）。
RateLimit	连接速率限制。属性：Enabled、MaxConnectionsPerIP（默认 10）、TimeWindowSec（默认 60）。
FloodProtection	消息洪水保护。属性：Enabled、MaxMessagesPerSec（默认 100）、Action（faDeny/faAllow/faLog）。
PayloadLimit	消息大小限制。属性：Enabled、MaxSizeBytes（默认 1048576）、Action（faDeny/faAllow/faLog）。
PathTraversal	路径遍历检测。属性：Enabled、Action（faDeny/faAllow/faLog）。
CommandInjection	命令注入检测。属性：Enabled、Action（faDeny/faAllow/faLog）、CustomPatterns（用于额外模式的 TStringList）。
ThreatScore	动态威胁评分。属性：Enabled、AutoBanThreshold（默认 80）、DecayPerHour（默认 5），以及用于微调评分计算的各违规类型权重。
BanEscalation	渐进式封禁升级。属性：Enabled、Levels（以秒为单位的持续时间 TStringList）、ResetAfterSec（默认 86400）。每次后续封禁使用下一级别的持续时间。级别值为 0 表示永久封禁。
GeoIP	地理 IP 过滤。属性：Enabled、Mode（gmBlockList/gmAllowList）、Countries（ISO 3166-1 alpha-2 国家代码的 TStringList）、DatabaseFile（GeoIP CSV 数据库的路径）。
BotDetection	基于 IP 的机器人检测。属性：Enabled、KnownBotsFile、CacheDurationSec（默认 3600）、DNSTimeoutMS（默认 2000），以及 ReverseDNS、Datacenter 和

	DNSBL 子对象。仅分类：将 IP 分类为已验证爬虫、数据中心主机、疑似机器人、列入黑名单或人工，而不进行阻止。
WebSocketProtection	WebSocket 特定保护。属性：Enabled、AllowedOrigins (TStringList)、MaxFrameSize (Integer)、AllowedSubprotocols (TStringList)。
CustomRules	自定义规则引擎。属性包括：Enabled、Rules (TsgcFirewallRuleItem 的 TCollection)。每条规则定义复杂过滤逻辑的条件和动作。

事件

OnFiltered：当连接或消息被阻止时触发。Allow 参数 (var Boolean) 允许您覆盖防火墙决策。

OnViolation：当检测到安全违规时触发。提供 IP 地址、违规类型和详细信息，用于日志记录。

OnResolveCountry：当 GeoIP 需要将 IP 解析为国家代码时触发。指定处理程序以提供自定义国家解析，或加载 GeoIP 数据库文件以进行自动查询。

OnBotDetected：当机器人检测对 IP 进行分类时触发。提供 IP 地址、TsgcBotClassification 判断以及匹配的机器人名称，用于日志记录或分析。

OnResolveBot：当机器人检测需要对 IP 进行分类时触发。分配处理程序以提供自定义判定结果，而不使用内置解析器。

OnThreatScoreChanged：当某个 IP 的威胁评分发生变化时触发。提供 IP 地址以及新旧评分值。

公开方法

方法	描述
IsConnectionAllowed(IP)	检查来自指定 IP 的连接是否被允许。与服务器集成时自动调用。
IsMessageAllowed(IP, Message)	检查来自给定 IP 的消息是否通过所有消息过滤器。与服务器集成时自动调用。
RegisterConnection(IP)	为速率限制跟踪注册新连接。自动调用。
UnregisterConnection(IP)	从跟踪中移除一个连接。在断开连接时自动调用。
RegisterFailedAttempt(IP)	记录一次失败的身份验证尝试，用于暴力破解检测。当身份验证失败时，从您的 OnAuthentication 事件处理程序中调用此方法。
BanIP(IP, DurationSec)	手动封禁一个 IP 地址。DurationSec = 0 表示永久封禁。
UnbanIP(IP)	移除对指定 IP 地址的封禁。
IsBanned(IP)	如果指定的 IP 当前被封禁，则返回 True。
ClearBans	删除所有活跃的封禁。

ClearTracking	重置所有内部跟踪数据（连接计数、尝试日志、消息计数）。
SaveBansToFile(FileName)	将所有活跃封禁记录保存到文件，以便在重启后持久保留。
LoadBansFromFile(FileName)	从之前保存的文件中恢复封禁记录。
LoadGeoIPDatabase(FileName)	加载 GeoIP CSV 数据库。预期格式：start_ip,end_ip,country_code。
LookupCountry(IP)	返回 IP 地址的两字母 ISO 国家代码。
GetBotClassification(IP)	返回某个 IP 地址的 TsgcBotClassification（缓存优先；缓存未命中时可能返回 bcUnknown 并触发异步查询）。
GetThreatScore(IP)	返回某 IP 地址的当前威胁分数（0-100）。
ResetThreatScore(IP)	将某个 IP 的威胁评分重置为 0。
GetTopThreats(Count)	返回威胁分数最高的 IP 列表。
IsOriginAllowed(Origin)	检查 WebSocket Origin 标头是否被 WebSocketProtection 配置允许。
IsFrameSizeAllowed(Size)	检查消息大小是否在 WebSocketProtection.MaxFrameSize 定义的限制范围内。
IsSubprotocolAllowed(Subprotocol)	检查 WebSocket 子协议是否被 WebSocketProtection 配置允许。
统计信息	公共只读 TsgcFirewallStats 对象，包含连接阻止数、消息过滤数、封禁数量、威胁评分和各模块命中计数的实时计数器。

自动集成

当分配给服务器的 Firewall 属性时，组件自动：

- **新连接时**：检查黑名单、白名单、封禁、速率限制、GeoIP 国家，并评估自定义规则。如果被阻止，则在 OnTCPConnect 事件之前拒绝连接。
- **收到消息时**：检查 SQL 注入、XSS 模式、载荷大小、路径穿越、命令注入、洪水限制，并评估自定义规则。更新威胁分数。若检测到违规，断开客户端连接。
- **断开连接时**：从内部追踪中注销连接。

CIDR 支持

黑名单和白名单支持 IP 范围的 CIDR 表示法：

- **192.168.1.0/24** — 匹配 192.168.1.0 到 192.168.1.255
- **10.0.0.0/8** — 匹配所有 10.x.x.x 地址
- **172.16.0.0/16** — 匹配 172.16.x.x 地址

示例

```
// Configure firewall
sgcWebSocketFirewall1->Blacklist->Enabled = true;
sgcWebSocketFirewall1->Blacklist->IPs->Add("10.0.0.0/8");
sgcWebSocketFirewall1->Whitelist->Enabled = true;
sgcWebSocketFirewall1->Whitelist->IPs->Add("192.168.1.1");
sgcWebSocketFirewall1->BruteForce->Enabled = true;
sgcWebSocketFirewall1->BruteForce->MaxAttempts = 3;
sgcWebSocketFirewall1->BruteForce->BanDurationSec = 600;
sgcWebSocketFirewall1->SQLInjection->Enabled = true;
sgcWebSocketFirewall1->XSS->Enabled = true;
sgcWebSocketFirewall1->RateLimit->Enabled = true;
sgcWebSocketFirewall1->RateLimit->MaxConnectionsPerIP = 5;
sgcWebSocketFirewall1->FloodProtection->Enabled = true;

// GeoIP: block connections from specific countries
sgcWebSocketFirewall1->GeoIP->Enabled = true;
sgcWebSocketFirewall1->GeoIP->Mode = gmBlockList;
sgcWebSocketFirewall1->GeoIP->Countries->Add("CN");
sgcWebSocketFirewall1->GeoIP->Countries->Add("RU");
sgcWebSocketFirewall1->LoadGeoIPDatabase("geoip.csv");

// Threat scoring with auto-ban
sgcWebSocketFirewall1->ThreatScore->Enabled = true;
sgcWebSocketFirewall1->ThreatScore->AutoBanThreshold = 80;

// Progressive ban escalation (5min, 30min, 2hr, 24hr, permanent)
sgcWebSocketFirewall1->BanEscalation->Enabled = true;
sgcWebSocketFirewall1->BanEscalation->Levels->Add("300");
sgcWebSocketFirewall1->BanEscalation->Levels->Add("1800");
sgcWebSocketFirewall1->BanEscalation->Levels->Add("7200");
sgcWebSocketFirewall1->BanEscalation->Levels->Add("86400");
sgcWebSocketFirewall1->BanEscalation->Levels->Add("0");

// Additional content protection
sgcWebSocketFirewall1->PayloadLimit->Enabled = true;
sgcWebSocketFirewall1->PayloadLimit->MaxSizeBytes = 65536;
sgcWebSocketFirewall1->PathTraversal->Enabled = true;
sgcWebSocketFirewall1->CommandInjection->Enabled = true;

// Persistent bans
sgcWebSocketFirewall1->SaveBansToFile("bans.dat");
sgcWebSocketFirewall1->LoadBansFromFile("bans.dat");

// Assign to server
sgcWebSocketHTTPServer1->Firewall = sgcWebSocketFirewall1;

// Optional: Track failed login attempts
void __fastcall TForm1::ServerAuthentication(TsgcWSConnection *Connection,
String aUser, String aPassword, bool &Authenticated)
{
    Authenticated = (aUser == "admin") && (aPassword == "secret");
    if (!Authenticated)
        sgcWebSocketFirewall1->RegisterFailedAttempt(Connection->IP);
}

// Optional: Handle firewall events for logging
void __fastcall TForm1::FirewallViolation(TObject *Sender, const String aIP,
const TsgcFirewallViolationType aViolationType, const String aDetails)
{
    Log("Firewall violation from " + aIP + ": " + aDetails);
}
```

检测到 SQL 注入模式

内置 SQL 注入检测器检查以下模式（不区分大小写）：

- 'OR', 'AND' — 布尔注入
- UNION SELECT — 基于联合的注入
- '; DROP, '; DELETE, '; INSERT, '; UPDATE — 语句注入
- -- (SQL 注释)
- EXEC(, EXECUTE(— 命令执行
- xp_cmdshell — 扩展存储过程
- CAST(, CONVERT(— 类型转换滥用
- 1=1, 1='1 — 恒真注入

可通过 `SQLInjection.CustomPatterns` 属性添加其他自定义模式。

检测到 XSS 模式

- <script — 脚本注入
- javascript: — 基于协议的 XSS
- onerror=, onload=, onclick=, onmouseover= — 事件处理程序注入
- eval() — JavaScript 执行
- document.cookie — Cookie 窃取
- <iframe, <object, <embed — 元素注入
- <svg onload — 基于 SVG 的 XSS
- expression() — CSS 表达式注入

检测到路径遍历模式

内置的路径遍历检测器检查以下模式：

- ../ 和 ..\ — 相对目录遍历
- %2e%2e%2f、%2e%2e/、..%2f——URL 编码变体
- %00 — 空字节注入
- /etc/passwd — Unix 敏感文件访问
- c:\windows — Windows 系统目录访问

检测到命令注入模式

内置命令注入检测器检查以下模式：

- ; | && || — Shell 操作符（命令链）
- `command` — 反引号执行
- \$(command) — 子 Shell 执行
- rm -rf — 破坏性文件操作
- wget, curl — 远程文件下载
- /bin/sh, /bin/bash — Unix shell 调用
- cmd.exe — Windows 命令解释器
- powershell — PowerShell 调用

可通过 `CommandInjection.CustomPatterns` 属性添加其他自定义模式。

线程安全

防火墙组件是完全线程安全的。所有公开方法均使用内部临界区来保护对跟踪数据的并发访问。已通过 20 个并发线程执行 100,000 次操作的压力测试，零错误、零内存泄漏。

该组件可在多个服务器实例之间安全共享，并可从任意线程（服务器事件处理程序、定时器线程等）访问，无需外部同步。

参考

- 属性 — 组件上提供 16 个属性。
- 方法 — 21 个公共方法。
- 事件——可处理 6 个事件。

防火墙：黑名单与白名单

TsgcWebSocketFirewall 通过两种互补机制提供基于 IP 的访问控制：黑名单（屏蔽特定 IP）和白名单（无条件放行受信任 IP）。

黑名单

黑名单阻止来自指定 IP 地址或 IP 范围的连接。启用后，任何来自列入黑名单 IP 的传入连接都会在到达服务器的连接事件之前被拒绝。

属性	描述
Blacklist.Enabled	启用或禁用黑名单检查。默认值：False。
Blacklist.IPs	包含已阻止 IP 地址或 CIDR 范围的 TStringList。

在设计时添加 IP

在对象检查器中单击 IPs 属性以打开字符串列表编辑器。每行添加一个 IP 或 CIDR 范围：

```
192.168.1.100
10.0.0.0/8
172.16.0.0/12
```

在运行时添加 IP

```
sgcWebSocketFirewall1->Blacklist->Enabled = true;
sgcWebSocketFirewall1->Blacklist->IPs->Add("192.168.1.100");
sgcWebSocketFirewall1->Blacklist->IPs->Add("10.0.0.0/8");
```

白名单

白名单无条件授予指定 IP 地址的访问权限。白名单中的 IP 绕过所有其他防火墙检查，包括黑名单、暴力破解封禁、频率限制和消息过滤。

属性	描述
Whitelist.Enabled	启用或禁用白名单检查。默认值：False。
Whitelist.IPs	包含受信任 IP 地址或 CIDR 范围的 TStringList。

示例

```
// 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");
```


优先顺序

当黑名单和白名单都已启用时，防火墙按以下顺序评估：

- 1. 若 IP 在白名单中，则**立即允许**连接。不再执行任何进一步检查。
- 2. 如果 IP 在黑名单中，则**拒绝**连接。
- 3. 如果 IP 不在任何列表中，则连接继续进行其他检查（暴力破解、速率限制等）。

这意味着对于同一 IP，白名单条目始终优先于黑名单条目。

CIDR 表示法

黑名单和白名单均支持 CIDR（无类域间路由）表示法来指定 IP 范围：

CIDR	Range	地址
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

您可以在同一列表中混合使用精确 IP 和 CIDR 范围：

```
sgcWebSocketFirewall1->Blacklist->IPs->Add("203.0.113.50"); // single IP
sgcWebSocketFirewall1->Blacklist->IPs->Add("198.51.100.0/24"); // entire subnet
```

黑名单和白名单的组合使用

一种常见模式是封锁一个宽泛的范围，但允许该范围内的特定 IP：

```
// 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");
```

在此示例中，10.x.x.x 范围内的所有 IP 均被封锁，但 10.1.1.50 例外，该地址在白名单中，可绕过所有检查。

防火墙：暴力破解保护

暴力破解保护模块检测来自同一 IP 地址的重复失败身份验证尝试，并自动封禁该 IP 一段可配置的时长。

工作原理

防火墙在滑动时间窗口内跟踪每个 IP 地址的失败尝试次数。当失败次数超过配置的阈值时，该 IP 将被自动封禁。被封禁的 IP 在连接级别被拒绝，在任何服务器事件触发之前。

1. 客户端连接并尝试身份验证。
2. 如果身份验证失败，您的代码将调用 **RegisterFailedAttempt(IP)** 通知防火墙。
3. 防火墙在当前时间窗口内增加该 IP 的失败计数器。
4. 如果计数器达到 **MaxAttempts**，则该 IP 将被封禁 **BanDurationSec** 秒。
5. 来自被封禁 IP 的任何新连接都会被立即拒绝。
6. 封禁时长到期后，IP 将自动解封，失败计数器重置。

属性

属性	默认	描述
BruteForce.Enabled	False	启用或禁用暴力破解保护。
BruteForce.MaxAttempts	5	在 IP 被封禁之前，时间窗口内允许的最大失败尝试次数。
BruteForce.TimeWindowSec	60	滑动时间窗口的持续时间（秒）。超过此时间的失败尝试将被丢弃。
BruteForce.BanDurationSec	300	自动封禁的持续时间（秒）。设为 0 表示永久封禁（直到手动移除或服务器重启）。

RegisterFailedAttempt

防火墙不知道哪些连接代表登录失败。您必须在身份验证事件处理程序中，在每次登录尝试失败时调用 **RegisterFailedAttempt**：

```
void __fastcall TForm1::ServerAuthentication(TsgcWSConnection *Connection,
String aUser, String aPassword, bool &Authenticated)
{
    Authenticated = CheckCredentials(aUser, aPassword);
    if (!Authenticated)
        sgcWebSocketFirewall1->RegisterFailedAttempt(Connection->IP);
}
```

手动封禁管理

除自动封禁外，您还可以手动管理封禁：

方法	描述
BanIP(IP, DurationSec)	手动封禁一个 IP。使用 DurationSec = 0 可实施永久封禁。
UnbanIP(IP)	从指定 IP 移除封禁。

IsBanned(IP)	如果 IP 当前被封禁，则返回 True。
ClearBans	移除所有活跃的封禁（包括自动封禁和手动封禁）。

示例：手动封禁

```
// Ban an IP for 1 hour
sgcWebsocketFirewall1->BanIP("203.0.113.50", 3600);

// Permanent ban
sgcWebsocketFirewall1->BanIP("198.51.100.25", 0);

// Check and unban
if (sgcWebsocketFirewall1->IsBanned("203.0.113.50"))
    sgcWebsocketFirewall1->UnbanIP("203.0.113.50");
```

配置示例

```
// 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;
```

注意事项

- 列入白名单的 IP 永远不会被封禁，即使对其调用 RegisterFailedAttempt 也是如此。
- 封禁记录默认存储在内存中。使用 **SaveBansToFile** 和 **LoadBansFromFile** 可在服务器重启后持久保留封禁记录。
- 时间窗口是滑动的，而非固定的。每次失败尝试都被单独追踪，并在 TimeWindowSec 秒后过期。

防火墙：SQL 注入和 XSS 检测

TsgcWebSocketFirewall 内置了针对 WebSocket 消息中 SQL 注入和跨站脚本（XSS）攻击的基于模式的检测功能。这些模块扫描传入消息内容，并在检测到威胁模式时采取相应措施。

SQL 注入检测

SQL 注入模块扫描所有传入的 WebSocket 消息中的常见 SQL 注入模式。检测不区分大小写。

属性

属性	描述
SQLInjection.Enabled	启用或禁用 SQL 注入检测。默认值：False。
SQLInjection.Action	检测到模式时采取的操作： faDeny （阻止消息并断开连接，默认）， faLog （触发 OnViolation 事件但允许消息）， faAllow （无操作，检测已禁用）。
SQLInjection.CustomPatterns	包含附加正则表达式模式的 TStringList，用于检查。这些模式在内置模式之外额外评估。

内置模式

默认情况下检查以下模式（不区分大小写）：

Pattern	攻击类型
' OR ', ' AND '	基于布尔值的注入
UNION SELECT	基于 Union 的注入
'; DROP, '; DELETE, '; INSERT, '; UPDATE	语句注入/数据操纵
--（双破折号）	SQL 注释注入
EXEC(, EXECUTE(	命令执行
xp_cmdshell	扩展存储过程滥用
CAST(, CONVERT(	类型转换滥用
1=1, 1='1	恒真式注入

自定义模式

您可以使用 CustomPatterns 属性添加特定于应用程序的模式：

```
sgcWebSocketFirewall1->SQLInjection->Enabled = true;
sgcWebSocketFirewall1->SQLInjection->CustomPatterns->Add("WAITFOR DELAY");
sgcWebSocketFirewall1->SQLInjection->CustomPatterns->Add("BENCHMARK\\(");
```

```
sgcWebSocketFirewall1->SQLInjection->CustomPatterns->Add("SLEEP\\(");
```

XSS 检测

XSS 模块检测 WebSocket 消息中的跨站脚本模式，防止注入可能被其他已连接客户端渲染的恶意脚本。

属性

属性	描述
XSS.Enabled	启用或禁用 XSS 检测。默认值：False。
XSS.Action	检测到模式时采取的动作： faDeny （阻止并断开连接，默认）、 faLog （触发 OnViolation 但允许）、 faAllow （不采取任何动作）。

内置模式

检测到以下 XSS 模式（不区分大小写）：

Pattern	攻击类型
<script	Script 标签注入
javascript:	基于协议的 XSS
onerror=, onload=, onclick=, onmouseover=	事件处理程序注入
eval(	JavaScript 代码求值
document.cookie	Cookie 窃取
<iframe, <object, <embed	元素注入
<svg onload	基于 SVG 的 XSS
expression(	CSS 表达式注入

Action 属性

SQLInjection 和 XSS 模块均支持类型为 TsgcFirewallAction 的 Action 属性：

Value	行为
faDeny	消息被阻止且客户端被断开连接。OnViolation 事件触发。这是默认行为。
faLog	消息被允许通过，但 OnViolation 事件会触发以供日志记录。
faAllow	不采取任何操作。在保持 Enabled 标志为 True 的同时，有效禁用检测。

OnViolation 事件

当检测到 SQL 注入或 XSS 模式时（Action 设置为 faDeny 或 faLog），OnViolation 事件以违规详情触发：

```
void __fastcall TForm1::FirewallViolation(TObject *Sender, const String aIP,
const TsgcFirewallViolationType aViolationType, const String aDetails)
{
    switch (aViolationType)
    {
        case fvSQLInjection:
            LogWarning("SQL injection attempt from " + aIP + ": " + aDetails);
            break;
        case fvXSS:
            LogWarning("XSS attempt from " + aIP + ": " + aDetails);
            break;
    }
}
```

配置示例

```
// 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;
```

注意事项

- 模式检测应用于所有传入的文本消息。二进制消息不被扫描。
- 白名单中的 IP 完全绕过消息过滤。
- 内置模式涵盖了最常见的攻击向量。对于特定于应用程序的威胁，请通过 SQLInjection.CustomPatterns 添加自定义模式。
- 在初始部署期间使用 faLog 模式监控检测结果而不影响客户端，在验证了应用程序的模式后切换到 faDeny。

防火墙：速率限制与洪水防护

TsgcWebSocketFirewall 提供两种互补的流量控制机制：连接速率限制（控制一个 IP 可以打开多少连接）和消息洪水保护（控制已连接客户端每秒可以发送多少消息）。

连接速率限制

速率限制约束单个 IP 地址在时间窗口内可建立的连接数，防止单个客户端耗尽所有可用的服务器连接。

属性

属性	默认	描述
RateLimit.Enabled	False	启用或禁用连接速率限制。
RateLimit.MaxConnectionsPerIP	10	单个 IP 地址允许的最大并发连接数。
RateLimit.TimeWindowSec	60	用于跟踪连接的时间窗口（秒）。超出此时间的连接将从计数中移除。

工作原理

1. 当有新连接到达时，防火墙统计来自该 IP 的活跃连接数量。
2. 如果计数达到或超过 MaxConnectionsPerIP，则新连接被拒绝。
3. 当连接断开时，将从跟踪计数中移除。
4. 超过 TimeWindowSec 的连接将自动从跟踪记录中清除。

示例

```
// Allow maximum 5 connections per IP
sgcWebSocketFirewall1->RateLimit->Enabled = true;
sgcWebSocketFirewall1->RateLimit->MaxConnectionsPerIP = 5;
sgcWebSocketFirewall1->RateLimit->TimeWindowSec = 60;
```

消息洪水防护

洪水保护限制单个 IP 每秒可发送的消息数量。这可防止滥用客户端通过快速消息突发压垮服务器。

属性

属性	默认	描述
FloodProtection.Enabled	False	启用或禁用消息洪泛保护。
FloodProtection.MaxMessagesPerSec	100	单个 IP 地址每秒允许的最大消息数量。
FloodProtection.Action	faDeny	超出限制时的动作： faDeny （断开客户端连接）、 faLog （触发 OnViolation 但允许通过）、 faAllow （不采取任何动作）。

工作原理

1. 防火墙为每个 IP 维护一个每秒重置的消息计数器。
2. 每条传入消息都会增加发送方 IP 的计数器。
3. 当计数器超过 MaxMessagesPerSec 时，执行已配置的 Action。
4. 使用 faDeny 动作时，客户端将立即断开连接。使用 faLog 时，OnViolation 事件触发，但消息正常处理。

示例

```
// Limit to 50 messages per second per IP
sgcWebSocketFirewall1->FloodProtection->Enabled = true;
sgcWebSocketFirewall1->FloodProtection->MaxMessagesPerSec = 50;
sgcWebSocketFirewall1->FloodProtection->Action = faDeny;
```

同时使用两种保护

速率限制和洪水保护协同工作，提供全面的流量控制：

```
// 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;
```

使用此配置，每个 IP 最多可以打开 5 个并发连接，在所有这些连接上，每秒最多可以发送合计 50 条消息。

调优指南

场景	MaxConnectionsPerIP	MaxMessagesPerSec
聊天应用程序	3-5	10-30
实时数据流	2-3	100-500
游戏服务器	1-2	50-100
API 网关	10-20	200-1000

从保守的限制开始，根据监控情况逐步提高。最初使用 faLog 模式观察流量模式，再启用 faDeny。

注意事项

- 白名单中的 IP 绕过速率限制和洪水保护。
- 速率限制追踪数据在连接关闭时自动清除。使用 ClearTracking 可手动重置所有计数器。
- 在反向代理后面，所有连接可能看起来都来自代理 IP。配置您的代理以传递真实的客户端 IP（例如 X-Forwarded-For 标头），并确保服务器正确解析它。

防火墙：Bot 检测

TsgcWebSocketFirewall 可将连接 IP 地址分类为机器人、爬虫、数据中心主机或人工访客。机器人检测结合了已知机器人数据库、正向确认反向 DNS 验证、数据中心/ASN 范围匹配和 DNSBL 封锁名单查找，为每个 IP 分配分类。这是一个**仅分类**模块：它本身不会阻止连接，而是报告判决结果，由应用程序决定如何处理。

工作原理

1. 一个 IP 连接到服务器。
2. 防火墙在结果缓存中查询该 IP。如果存在缓存分类（且未过期），则立即使用。
3. 缓存未命中时，将 IP 与已知机器人文件（已验证的爬虫）和数据中心 ASN 文件（云/托管范围）进行匹配。
4. 如果启用了 ReverseDNS 或 DNSBL 验证，则会在后台线程上安排查找，以免阻塞接受线程。在查找完成之前，IP 可能被归类为 **bcUnknown**。
5. 分类解析完成后，结果将存储在缓存中（持续 CacheDurationSec 秒），并触发 **OnBotDetected** 事件。
6. 分类结果将更新防火墙统计信息（fvBot 违规计数器），并可通过规则的 **BotType** 属性供自定义规则使用。

属性

属性	描述
BotDetection.Enabled	启用或禁用机器人检测。默认值：False。
BotDetection.KnownBotsFile	经过验证的爬虫 CSV 文件路径。每行格式为 CIDR,botname （例如 IP 范围及其所属爬虫）。
BotDetection.CacheDurationSec	已解析分类在结果缓存中保留的秒数，超过后将重新查询。默认：3600。
BotDetection.DNSTimeoutMS	每次 DNS 查询（反向 DNS 和 DNSBL）的超时时间（毫秒）。默认值：2000。
BotDetection.ReverseDNS.Enabled	启用声明爬虫的正向确认反向 DNS（FCrDNS）验证。默认值：False。
BotDetection.Datacenter.Enabled	启用数据中心/云托管范围检测。默认值：False。
BotDetection.Datacenter.ASNFile	数据中心 ASN 范围 CSV 文件路径。每行格式为 CIDR,asname （例如云服务商 IP 段及其 ASN 名称）。
BotDetection.DNSBL.Enabled	启用 DNSBL（DNS 阻止列表）查找。默认值：False。
BotDetection.DNSBL.Zones	用于查询的 DNSBL 区域 TStringList（例如 zen.spamhaus.org ）。

分类值

TsgcBotClassification 枚举描述分配给 IP 的判决：

Value	描述
bcUnknown	该 IP 尚未完成分类。在异步查找进行中时，缓存未命中时返回此值。

bcVerifiedCrawler	经已知机器人文件或正向确认反向 DNS 确认的已知合法爬虫（例如搜索引擎机器人）。
bcDatacenter	该 IP 属于 ASN 文件中列出的云或托管服务提供商地址范围。通常为自动化流量，而非住宅用户。
bcSuspectedBot	该 IP 表现出机器人特征，但无法与已验证的爬虫匹配。
bcBlocklisted	该 IP 列在某个已配置的 DNSBL 区域中。
bcHuman	该 IP 看起来像普通的人工访问者（无机器人、数据中心或黑名单匹配）。

CSV 文件格式

已知机器人文件（**KnownBotsFile**）列出已验证的爬虫范围。每行是一个 CIDR 范围，后跟机器人名称：

```
66.249.64.0/19,Googlebot
157.55.0.0/16,Bingbot
17.58.0.0/16,Applebot
```

数据中心 ASN 文件（**Datacenter.ASNFile**）列出云/托管范围。每行是一个 CIDR 范围，后跟 ASN 名称：

```
3.0.0.0/15,AMAZON-AES
13.64.0.0/11,MICROSOFT-AZURE
34.64.0.0/10,GOOGLE-CLOUD
```

启用机器人检测

启用模块，加载已知机器人和 ASN 文件，并开启您需要的验证子模块。DNS 查找在后台线程上运行，因此启用 ReverseDNS 或 DNSBL 不会减慢接受线程的速度。

```
// Enable bot detection
sgcWebsocketFirewall1->BotDetection->Enabled = true;
sgcWebsocketFirewall1->BotDetection->KnownBotsFile = "C:\\data\\known_bots.csv";
sgcWebsocketFirewall1->BotDetection->CacheDurationSec = 3600;
sgcWebsocketFirewall1->BotDetection->DNSTimeoutMS = 2000;

// Forward-confirmed reverse DNS verification
sgcWebsocketFirewall1->BotDetection->ReverseDNS->Enabled = true;

// Datacenter / cloud-hosting detection
sgcWebsocketFirewall1->BotDetection->Datacenter->Enabled = true;
sgcWebsocketFirewall1->BotDetection->Datacenter->ASNFile = "C:\\data\\datacenter_asn.csv";

// DNSBL blocklist lookups
sgcWebsocketFirewall1->BotDetection->DNSBL->Enabled = true;
sgcWebsocketFirewall1->BotDetection->DNSBL->Zones->Add("zen.spamhaus.org");
```

仅分类行为与 OnBotDetected

Bot 检测不会自行拒绝连接。当某个 IP 被分类后，将触发 **OnBotDetected** 事件，更新防火墙统计信息（fvBot 计数器）并将结果存储在缓存中。使用该事件可记录流量、构建分析或接入您自己的逻辑：

```
void __fastcall TForm1::FirewallBotDetected(TObject *Sender, const String aIP,
TsgcBotClassification aClassification, const String aBotName)
{
    switch (aClassification)
    {
        case bcVerifiedCrawler: Log("Verified crawler " + aBotName + " from " + aIP); break;
        case bcDatacenter:     Log("Datacenter host from " + aIP); break;
        case bcBlocklisted:    Log("Blocklisted IP " + aIP); break;
    }
}
```

```
}
```

通过自定义规则对分类结果采取行动

由于机器人检测仅进行分类，拦截或封禁操作通过自定义规则引擎完成。每个 **TsgcFirewallRuleItem** 都有一个 **BotType** 属性：设置后，该规则将匹配分类等于该值的连接，然后您可以对其执行拒绝、封禁或日志记录操作。

```
sgcWebsocketFirewall1->CustomRules->Enabled = true;

// Deny connections coming from blocklisted IPs
TsgcFirewallRuleItem *vRule =
    (TsgcFirewallRuleItem *)sgcWebsocketFirewall1->CustomRules->Rules->Add();
vRule->BotType = bcBlocklisted;
vRule->ActionType = raDeny;
```

使用 OnResolveBot 自定义裁决

如果您拥有自己的机器人检测逻辑（第三方信誉服务、内部允许/拒绝列表等），请分配 **OnResolveBot** 事件。分配后，您的处理程序将提供判断结果，取代内置的解析器。通过设置 *aClassification* 和 *aBotName* 变量参数来返回您的结果：

```
void __fastcall TForm1::FirewallResolveBot(TObject *Sender, const String aIP,
    TsgcBotClassification &aClassification, String &aBotName)
{
    // Custom lookup - query your own bot intelligence service
    if (MyBotService->IsCrawler(aIP))
    {
        aClassification = bcVerifiedCrawler;
        aBotName = MyBotService->GetName(aIP);
    }
    else
        aClassification = bcHuman;
}
```

注意：当分配了 **OnResolveBot** 时，该 IP 的内置解析器（已知机器人文件、反向 DNS、数据中心和 DNSBL 检查）将被绕过。您的处理程序返回的裁决会被缓存，并像内置结果一样传递给 **OnBotDetected** 和自定义规则。

TsgcWSRateLimiter

TsgcWSRateLimiter 实现了一个功能完整的速率限制组件，可保护服务器端点免受过量流量、滥用和抓取的影响。

简介

TsgcWSRateLimiter 实现了一个全功能的速率限制组件，保护服务器端点免受过度流量、滥用和抓取的影响。它支持三种算法（令牌桶、滑动窗口、固定窗口）、作用域规则（按 IP、按 API 密钥、按用户、按端点）、长期配额（小时/天/月）、突发保护、可配置的 HTTP 429 响应格式和实时统计信息。内部状态可以持久化到文件或流，以便计数器在重启后依然有效。

速率限制器通过 TsgcWebSocketHTTPServer 和 TsgcWSServer_HTTPAPI 上提供的 **RateLimiter** 属性自动与服务器组件集成。一旦赋值，无需手动绑定事件即可评估每个连接和每条消息。

快速入门

1. 将 **TsgcWSRateLimiter** 组件拖放到窗体上。
2. 启用一个或多个策略和范围（请参阅下方各节文档）。
3. 将速率限制器分配给服务器组件：

```
sgcWebSocketHTTPServer1.RateLimiter := sgcWSRateLimiter1;
```

4. 启动服务器。速率限制器自动评估每个连接和消息，以 HTTP 429 拒绝超额流量并更新统计计数器。

顶级属性

功能说明： 组件的主开关，以及状态持久化方法使用的可选存储文件。当 Enabled 为 False 时，所有公共方法均短路为"允许"状态，且不更新任何计数器。

使用场景： 在受控负载测试或维护窗口期间，在运行时禁用速率限制器，而无需将其从服务器中移除。设置 StorageFile 以使计数器在应用程序重启后依然存在，例如滚动式令牌桶将从上次停止的位置继续填充，而不是从头开始并接受一次大量突发请求。

属性	类型	默认	描述
已启用	Boolean	True	主开关。当为 False 时，IsAllowed / Consume / IsConnectionAllowed / IsMessageAllowed 始终返回 True，且不更新任何计数器。
StorageFile	string	"	可选文件路径，供 SaveStateToFile / LoadStateFromFile 用于在重启后持久化计数器。

```
sgcWSRateLimiter1->Enabled = true;
sgcWSRateLimiter1->StorageFile = "ratelimit.dat";
sgcWSRateLimiter1->LoadStateFromFile(sgcWSRateLimiter1->StorageFile);
```

TokenBucket 策略

功能：为每个密钥维护一个最多 *Capacity* 个令牌的桶。桶每 *RefillIntervalMs* 毫秒以 *RefillRate* 个令牌的速率重新填充。每个请求消耗一个令牌（或传递给 *Consume* 的 *Cost* 参数）。当桶为空时，请求被拒绝，并根据重新填充计划计算 *RetryAfterSec* 提示。

使用场景：API 公告平均速率为每秒 10 个请求，但您希望让行为良好的客户端能够连续突发发送最多 50 个请求而不被限流。令牌桶（*Capacity*=50，*RefillRate* 为每 1000 毫秒 10 个）允许客户端即时发送 50 个请求，然后将其限制为每秒 10 个，直到桶重新填满。

属性	类型	默认	描述
已启用	Boolean	False	将令牌桶算法启用为全局默认策略。
容量	Integer	100	令牌桶可持有的最大令牌数量，控制最大突发大小。
RefillRate	Integer	10	每次 <i>RefillIntervalMs</i> 向桶中添加的令牌数量。
RefillIntervalMs	Integer	1000	补充周期（毫秒）。与 <i>RefillRate</i> 共同决定持续请求速率。

```
// 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 策略

工作原理：为每个请求存储一个时间戳，并统计落在 *WindowSec* 秒滚动窗口内的时间戳数量。当计数将超过 *MaxRequests* 时拒绝。窗口持续滑动，因此不存在突发流量可以溜过的硬重置时刻。

使用场景：您发布了严格的 SLA，保证“在任意 60 秒滚动窗口内不超过 100 个请求”。固定窗口会让客户端在 12:00:59 发送 100 个请求，在 12:01:00 再发送 100 个请求（在一秒内共 200 个）。滑动窗口可防止这种情况并精确执行 SLA。

属性	类型	默认	描述
已启用	Boolean	False	将滑动窗口算法启用为全局默认策略。
WindowSec	Integer	60	滚动窗口的大小（秒）。
MaxRequests	Integer	1000	在任意 <i>WindowSec</i> 宽度的滚动窗口内允许的最大请求数。

```
// 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
```

固定窗口策略

功能说明：在 *WindowSec* 秒的固定时钟窗口内计数请求，并在窗口边界重置计数器为零。这是最简单的算法，内存开销也最低；但在边界附近的短时间内最多可允许 $2 \times \text{MaxRequests}$ 个请求。

何时使用：类似账单的配额，如“每小时 1000 次请求，计数器在整点重置”。客户期望有清晰可见的重置时间，而轻微的边界突发行为是可接受的，因为计费单位是小时而不是滚动分钟。

属性	类型	默认	描述
已启用	Boolean	False	将固定窗口算法启用为全局默认策略。
WindowSec	Integer	60	固定窗口的大小，单位为秒。窗口结束时，计数器重置为 0。
MaxRequests	Integer	1000	单个固定窗口内允许的最大请求数。

```
// Billing-style: 1000 requests per hour, resets on the hour
sgcWSRateLimiter1->FixedWindow->Enabled = true;
sgcWSRateLimiter1->FixedWindow->WindowSec = 3600;
sgcWSRateLimiter1->FixedWindow->MaxRequests = 1000;
```

PerIP

功能说明：定义限定于来源 IP 地址的速率限制规则。当请求键与 *PerAPIKey* / *PerUser* / *PerEndpoint* 均不匹配时，*PerIP* 成为活跃规则并覆盖全局的 *Token/Sliding/Fixed* 默认值。每个 IP 拥有独立的计数器。

使用场景：拥有众多客户端的公共 API。若不使用 *PerIP*，单个嘈杂客户端可能耗尽全局配额并影响其他所有人。为每个 IP 分配独立的配额桶（例如每分钟 100 次请求），这样一个客户端就不会影响其他客户端。

属性	类型	默认	描述
已启用	Boolean	False	启用每 IP 速率限制。
MaxRequests	Integer	100	每个窗口的最大请求数（或令牌桶策略时的桶容量）。
WindowSec	Integer	60	窗口大小（秒）。对于令牌桶，也用于推导补充速率（ $\text{MaxRequests} / \text{WindowSec}$ 个令牌/秒）。

策略	TsgcRateLimitStrategy	rlsTokenBucket	此范围使用的算法：rlsTokenBucket、rlsSliding 或 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

功能说明：语义与 PerIP 相同，但作用域限定为 API 密钥标识符。组件将以前缀 `apikey:` 开头的密钥识别为 API 密钥作用域。设计用于与 TsgcWSAPIKeyManager 集成，使每个客户获得独立的速率预算，与来源 IP 无关。

适用场景：采用分层定价的 SaaS。每个客户都有自己的 API 密钥和独立的配额，与连接来源无关。将 PerAPIKey 设为默认层，并通过按路径前缀键控的 PerEndpoint 规则，或在 OnThrottled 事件处理程序内覆盖决策来区分客户层（Free / Pro / Enterprise）。

属性	类型	默认	描述
已启用	Boolean	False	启用按 API 密钥的速率限制。
MaxRequests	Integer	100	每个时间窗口的最大请求数（或令牌桶的桶容量）。
WindowSec	Integer	60	窗口大小（秒）。
策略	TsgcRateLimitStrategy	rlsTokenBucket	此作用域所用的算法。

```
// 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

功能说明：与 PerIP/PerAPIKey 结构相同，但以应用程序定义的用户标识符为键。组件将以 `user:` 前缀开头的键识别为用户范围。当单个用户可能从多个设备或 IP 连接，且您希望该用户拥有单一共享配额时，此功能非常有用。

适用场景：已认证的 Web 或移动应用。用户 `alice@example.com` 每天允许 1000 次请求，无论她从手机、笔记本电脑还是工作桌面访问。按 IP 聚合会重复计数，按用户聚合才能得到实际计费所需的配额。

属性	类型	默认	描述
已启用	Boolean	False	启用每用户速率限制。
MaxRequests	Integer	100	每个时间窗口的最大请求数（或令牌桶的桶容量）。
WindowSec	Integer	60	窗口大小（秒）。
策略	TsgcRateLimitStrategy	rlsTokenBucket	此作用域所用的算法。

```
// 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

功能说明：基于模式的规则集合。每个条目都有自己的 *Pattern*（支持通配符 * 和 ?）、*Strategy*、*MaxRequests* 和 *WindowSec*。第一个模式匹配的已启用条目获胜——不再评估后续条目。优先级高于 PerAPIKey、PerUser 和 PerIP。

使用场景：您有一个需要按客户端限制为每分钟 10 次请求的高代价端点 `/api/v1/expense-report`，而廉价的状态端点 `/api/v1/status` 允许每分钟 1000 次请求。不同的端点有不同的配额，请为每个路径模式配置一个 PerEndpoint 条目。

属性	类型	默认	描述
已启用	Boolean	False	启用 PerEndpoint 规则集。当为 False 时，不评估任何条目。
规则	TCollection	空	TsgcRateLimitRuleItem 的集合。顺序很重要，首个匹配规则优先。

每个 **TsgcRateLimitRuleItem** 公开以下内容：

属性	类型	默认	描述
名称	string	"	在对象检查器中显示的友好名称。
已启用	Boolean	True	禁用此条目而不删除它。
模式	string	"	与密钥匹配的通配符模式（不区分大小写）。支持 * 和 ?。
MaxRequests	Integer	100	每个窗口的最大请求数 / 桶容量。
WindowSec	Integer	60	窗口大小（秒）。

策略	TsgcRateLimitStrategy	rlsTokenBucket	用于匹配请求的算法。

```
sgcWSRateLimiter1->PerEndpoint->Enabled = true;

TsgcRateLimitRuleItem *ruleReport = (TsgcRateLimitRuleItem*)
    sgcWSRateLimiter1->PerEndpoint->Rules->Add();
ruleReport->Name = "expensive-report";
ruleReport->Pattern = "*/api/v1/expensive-report*";
ruleReport->Strategy = rlsSliding;
ruleReport->MaxRequests = 10;
ruleReport->WindowSec = 60;

TsgcRateLimitRuleItem *ruleStatus = (TsgcRateLimitRuleItem*)
    sgcWSRateLimiter1->PerEndpoint->Rules->Add();
ruleStatus->Name = "status";
ruleStatus->Pattern = "*/api/v1/status*";
ruleStatus->Strategy = rlsTokenBucket;
ruleStatus->MaxRequests = 1000;
ruleStatus->WindowSec = 60;
```

配额

功能描述：超出主策略滚动/补充窗口的长期上限。每个条目限制一个密钥（或整个系统）在某个日历周期（*qpHour*、*qpDay* 或 *qpMonth*）内可发出的请求数量。一旦达到 *Limit*，该请求将被拒绝，并触发 *OnQuotaExceeded* 事件。

使用场景：免费套餐计划每月硬性限制 10,000 次请求。滚动令牌桶仍可平滑短期流量，但当月配额耗尽后，直到下一个日历月前，所有后续调用均将失败。

属性	类型	默认	描述
已启用	Boolean	False	启用配额规则集。为 False 时不评估任何条目。
配额	TCollection	空	TsgcRateLimitQuotaltem 的集合。所有已启用的条目将按顺序进行评估。

每个 **TsgcRateLimitQuotaltem** 公开：

属性	类型	默认	描述
名称	string	"	在 <i>OnQuotaExceeded</i> 中及 <i>Result.Reason</i> ("quota:Name") 中报告的友好名称。
已启用	Boolean	True	禁用此配额但不删除它。
作用域	TsgcRateLimitQuotaScope	qscPerKey	qscPerKey 按密钥独立计数；qscGlobal 对所有密钥合并计数。
周期	TsgcRateLimitQuotaPeriod	qpHour	qpHour、qpDay 或 qpMonth。计数器在每个周期开始时自动重置。
限制	Int64	10000	在该时间段内允许的最大请求数。

```
sgcWSRateLimiter1->Quotas->Enabled = true;

TsgcRateLimitQuotaItem *quota = (TsgcRateLimitQuotaItem*)
    sgcWSRateLimiter1->Quotas->Quotas->Add();
quota->Name = "free-tier-monthly";
quota->Scope = qscPerKey;
quota->Period = qpMonth;
quota->Limit = 10000;
```

BurstProtection

功能说明：与主策略并行运行的短时间尺度尖峰检测器。如果来自同一密钥的请求在 *BurstWindowMs* 毫秒内超过 *BurstThreshold* 个，该密钥将进入 *CooldownSec* 秒的冷却期，期间所有请求都将被拒绝，即使主策略本会允许这些请求。

适用场景：检测在 100 毫秒内以 100 多个请求轰炸服务器但每分钟总量不超限的爬虫和扫描客户端。突发保护能捕获这种人工用户永远不会产生的气枪式请求模式。

属性	类型	默认	描述
已启用	Boolean	False	启用突发检测器。
BurstThreshold	Integer	50	在 BurstWindowMs 时间窗口内触发冷却期的请求数量。
BurstWindowMs	Integer	500	突发观察窗口（毫秒）。
CooldownSec	Integer	30	检测到突发后冷却期的持续时间。在冷却期间，每个请求都会被拒绝。

```
// 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;
```

响应

功能说明：配置当请求被限流时返回给客户端的 HTTP 响应格式。控制状态码、正文消息以及是否在响应中添加 *Retry-After* 和 *X-RateLimit-** 头。

使用场景：您希望行为良好的客户端正确地退避。返回标准 429 状态，并附带 *Retry-After* 标头以指示等待时长，同时包含 *X-RateLimit-Limit/X-RateLimit-Remaining/X-RateLimit-Reset*，使客户端能够主动调整速度，而不是对拒绝做出被动反应。

属性	类型	默认	描述

StatusCode	Integer	429	对被限速请求返回的 HTTP 状态码。使用 429（标准）或 503 用于维护式拒绝。
消息	string	"Too Many Requests"	被限速请求的 HTTP 响应主体。
RetryAfterHeader	Boolean	True	为 True 时，添加一个 <code>Retry-After</code> 头，其值为计算得出的重试延迟（秒）。
IncludeRateLimitHeaders	Boolean	True	为 True 时，向每个响应添加 <code>X-RateLimit-Limit</code> / <code>X-RateLimit-Remaining</code> / <code>X-RateLimit-Reset</code> 标头。

```
sgcWSRateLimiter1->Response->StatusCode = 429;
sgcWSRateLimiter1->Response->Message = "Too Many Requests - please slow down";
sgcWSRateLimiter1->Response->RetryAfterHeader = true;
sgcWSRateLimiter1->Response->IncludeRateLimitHeaders = true;
```

统计信息

功能：只读计数器对象，暴露有关速率限制器的实时统计信息。所有计数器在组件的内部临界区内原子更新。

使用场景：管理 UI 中的实时仪表板。从 TTimer 轮询 Stats，并在标签、仪表或图表中展示当前 TotalRequests/TotalThrottled/ActiveKeys，使运维人员可以实时看到流量压力。

属性	类型	描述
TotalRequests	Int64	自上次 Reset 以来评估的请求总次数。
TotalThrottled	Int64	被策略或突发保护拒绝的请求总数。
TotalQuotaExceeded	Int64	因达到配额上限而被拒绝的请求总数。
ActiveKeys	Integer	当前追踪的键数量（桶 + 窗口计数器 + 活动连接）。
正常运行时间	Int64	统计对象上次重置以来经过的秒数。

```
// Poll from a TTimer and update a dashboard
void __fastcall TForm1::StatsTimer(TObject *Sender)
{
    LabelRequests->Caption = "Requests: " + IntToStr(sgcWSRateLimiter1->Stats->TotalRequests);
    LabelThrottled->Caption = "Throttled: " + IntToStr(sgcWSRateLimiter1->Stats->TotalThrottled);
    LabelActiveKeys->Caption = "Active keys: " + IntToStr(sgcWSRateLimiter1->Stats->ActiveKeys);
    LabelUptime->Caption = "Uptime: " + IntToStr(sgcWSRateLimiter1->Stats->Uptime) + " s";
}
```

事件

OnThrottled：当请求被策略或突发保护拒绝时触发。提供密钥（IP、API 密钥或用户）和原因字符串（"token_bucket"、"sliding_window"、"fixed_window"、"burst"、"connection_rate"或"message_rate"）。*Allow* 参数（var Boolean）允许您覆盖决策并放行请求。

OnQuotaExceeded：当达到长期配额时触发。提供已超出的密钥和配额名称。

OnStateChange：当某个键的令牌桶大小发生变化时触发。提供该键以及桶大小的新旧值。适用于遥测、调试和实时可视化。

公开方法

方法	描述
IsAllowed(Key, Cost)	如果指定密钥的给定 Cost（默认为 1）的请求被允许，则返回 True。
Consume(Key, Cost)	尝试消耗 Cost 个令牌，并返回一个 TsgcRateLimitResult（Allowed、Remaining、RetryAfterSec、Reason）。
Reset(Key)	清除指定密钥的计数器。
ResetAll	清除每个被跟踪键的所有内部计数器，并重置 Stats 对象。
GetRemaining(Key)	返回指定密钥的剩余可用请求数。
GetRetryAfter(Key)	返回调用方在重试前应等待的秒数。
SaveStateToFile(FileName)	将所有桶和计数器持久化到给定文件。
LoadStateFromFile(FileName)	从之前保存到文件的数据恢复桶和计数器。
SaveStateToStream(Stream)	将内部状态持久化到任意流。
LoadStateFromStream(Stream)	从任意流中恢复内部状态。
IsConnectionAllowed(IP)	服务器集成钩子。如果来自 IP 的新连接通过速率限制，则返回 True。由服务器自动调用。
IsMessageAllowed(IP, Message)	服务器集成钩子。如果来自 IP 的消息通过速率限制，则返回 True。由服务器自动调用。
RegisterConnection(IP)	注册新连接以供追踪。此方法会自动调用。
UnregisterConnection(IP)	从跟踪中移除一个连接。在断开连接时自动调用。

自动集成

分配给服务器的 RateLimiter 属性后，该组件将自动：

- **新连接建立时**：调用 IsConnectionAllowed 以评估 PerIP、突发保护和全局策略。若超出限制，则在 OnConnect 事件之前通过 HTTP 429 拒绝连接。调用 RegisterConnection 以跟踪客户端。
- **收到消息时**：调用 IsMessageAllowed 以检查 PerIP、PerEndpoint、PerAPIKey、PerUser、配额和突发保护。超过限制时断开连接或拒绝消息。
- **断开连接时**：调用 UnregisterConnection 以释放跟踪资源。

完整示例

```
// Enable rate limiter
sgcWSRateLimiter1->Enabled = true;

// Global Token Bucket: 50 burst, 10/sec sustained
sgcWSRateLimiter1->TokenBucket->Enabled = true;
sgcWSRateLimiter1->TokenBucket->Capacity = 50;
sgcWSRateLimiter1->TokenBucket->RefillRate = 10;
sgcWSRateLimiter1->TokenBucket->RefillIntervalMs = 1000;

// Per-IP sliding window: 60 requests per minute per IP
sgcWSRateLimiter1->PerIP->Enabled = true;
sgcWSRateLimiter1->PerIP->Strategy = rlsSliding;
sgcWSRateLimiter1->PerIP->MaxRequests = 60;
sgcWSRateLimiter1->PerIP->WindowSec = 60;

// Per-endpoint: expensive report endpoint = 10 req/min
sgcWSRateLimiter1->PerEndpoint->Enabled = true;
TsgcRateLimitRuleItem *rule = (TsgcRateLimitRuleItem*)
    sgcWSRateLimiter1->PerEndpoint->Rules->Add();
rule->Name = "expensive-report";
rule->Pattern = "*/api/v1/expensive-report*";
rule->Strategy = rlsSliding;
rule->MaxRequests = 10;
rule->WindowSec = 60;

// Burst protection: 50 requests in 500ms triggers 30s cooldown
sgcWSRateLimiter1->BurstProtection->Enabled = true;
sgcWSRateLimiter1->BurstProtection->BurstThreshold = 50;
sgcWSRateLimiter1->BurstProtection->BurstWindowMs = 500;
sgcWSRateLimiter1->BurstProtection->CooldownSec = 30;

// Monthly quota per API key: 10000 requests/month
sgcWSRateLimiter1->Quotas->Enabled = true;
TsgcRateLimitQuotaItem *quota = (TsgcRateLimitQuotaItem*)
    sgcWSRateLimiter1->Quotas->Quotas->Add();
quota->Name = "free-tier-monthly";
quota->Scope = qscPerKey;
quota->Period = qpMonth;
quota->Limit = 10000;

// HTTP 429 response with Retry-After and X-RateLimit-* headers
sgcWSRateLimiter1->Response->StatusCode = 429;
sgcWSRateLimiter1->Response->Message = "Too Many Requests";
sgcWSRateLimiter1->Response->RetryAfterHeader = true;
sgcWSRateLimiter1->Response->IncludeRateLimitHeaders = true;

// Persist state across restarts
sgcWSRateLimiter1->StorageFile = "ratelimit.dat";
sgcWSRateLimiter1->LoadStateFromFile("ratelimit.dat");

// Assign to server
sgcWebSocketHTTPServer1->RateLimiter = sgcWSRateLimiter1;

// Handle throttled events
void __fastcall TForm1::RateLimiterThrottled(TObject *Sender,
    const String aKey, const String aReason, bool &Allow)
{
    Log("Throttled " + aKey + ": " + aReason);
}

// Handle quota exhausted events
void __fastcall TForm1::RateLimiterQuotaExceeded(TObject *Sender,
    const String aKey, const String aQuotaName)
{
    Log("Quota " + aQuotaName + " exceeded for " + aKey);
}
```

线程安全

限速器完全线程安全。所有公共方法使用内部临界区和线程安全计数器来保护并发访问。该组件可由多个服务器实例共享，并可从任何线程（服务器事件处理程序、定时器线程、后台工作线程）访问，无需外部同步。

参考

- 属性 — 组件上提供 13 个属性。
- 方法 — 12 个公共方法。
- 事件 — 3 个可处理的事件。

TsgcWSCircuitBreaker

TsgcWSCircuitBreaker 为客户端调用 HTTP API 实现断路器弹性模式。

简介

TsgcWSCircuitBreaker 为客户端调用 HTTP API 实现了断路器弹性模式。当远程端点不可用时，它通过在滚动窗口中跟踪失败、在达到可配置阈值后打开断路器、并在冷却期间立即拒绝调用，防止级联故障。它支持失败次数和失败率阈值、慢速调用检测、半开试探调用、异常分类（记录/忽略）、按端点覆盖、回退响应以及实时指标。

断路器通过 **CircuitBreaker** 属性自动与 **TsgcHTTPAPI_client** 的所有子类（OpenAI、Anthropic、Gemini、Grok、DeepSeek、Mistral、Ollama、Stripe、Binance、Bybit、Coinbase、Kraken 及其他 API 包装器）集成。一经赋值，每个 Get / Post / Put / Patch / Delete 调用都会自动被包装。每个唯一主机名维护一个独立的断路器。

快速入门

1.在表单上放置一个 **TsgcWSCircuitBreaker** 组件。

2. 配置阈值（何时打开）和恢复（如何恢复）：

```
sgcWSCircuitBreaker1.Thresholds.FailureCount := 5;
sgcWSCircuitBreaker1.Recovery.CooldownSec := 30;
```

3. 将断路器分配给 HTTP API 客户端（任何 TsgcHTTPAPI_client 的子类）：

```
sgcAI_OpenAI1.CircuitBreaker := sgcWSCircuitBreaker1;
```

4. 在客户端上调用 Get/Post/Delete/Patch/Put。断路器会自动记录成功和失败，并在断路器 Open 时拒绝新的调用。

状态

功能说明：断路器是一个三状态机。每个主机（键）拥有独立的状态，控制出站调用是被允许、拒绝还是作为试探性请求发送。

State	行为	Transition
csClosed	所有调用均通过。失败计入滚动窗口。	当达到 FailureCount、FailureRatePercent 或 SlowCallRatePercent 时，移至 csOpen 状态。
csOpen	每次调用都被立即拒绝。每次尝试都会触发 OnCallRejected。	在距开路时刻经过 Recovery.CooldownSec 秒后，转移至 csHalfOpen 状态。
csHalfOpen	允许最多 Recovery.HalfOpenTrialCalls 次试验调用以探测恢复状态。	当试验成功时转为 csClosed；当任何试验失败时退回到 csOpen。

顶级属性

功能说明： 组件的主开关，以及在未提供每次调用的密钥时使用的两个默认密钥名称。客户端集成自动使用目标主机名作为密钥，DefaultKey 和 ServerKey 仅适用于手动调用 Execute/RecordSuccess/RecordFailure 以及可选的服务器集成辅助方法。

使用时机： 在受控负载测试或维护窗口期间在运行时禁用熔断器，而无需取消其与 HTTP 客户端的关联。当您在自定义代码内手动驱动 Execute 时，请设置一个有意义的 DefaultKey（例如 'openai-api'）。

属性	类型	默认	描述
已启用	Boolean	True	主开关。当为 False 时，每个公共方法都会短路为"允许"，且不更新任何计数器。
DefaultKey	string	'default'	无参数重载的 Execute、RecordSuccess 和 RecordFailure 使用的密钥。
ServerKey	string	'server'	供可选服务器集成钩子（IsConnectionAllowed、IsMessageAllowed、RecordMessageError）使用的密钥。

```
sgcWSCircuitBreaker1->Enabled = true;  
sgcWSCircuitBreaker1->DefaultKey = "openai-api";  
sgcWSCircuitBreaker1->ServerKey = "server";
```

阈值

功能说明： 定义使断路器从 Closed 变为 Open 的条件。当达到 FailureCount 个原始失败次数，或窗口内 FailureRatePercent 的调用失败（超过 MinCalls 后），或 SlowCallRatePercent 的调用超过 SlowCallDurationMs 时，断路器将打开。

使用场景： 您的服务器调用 OpenAI 的 API。当 60 秒内出现 5 次失败（或在观察到至少 10 次调用后有 50% 的调用失败），则断开电路并停止轰击 OpenAI，让远端自行恢复，而不是继续堆积重试。

属性	类型	默认	描述
已启用	Boolean	True	启用基于阈值的开路功能。当值为 False 时，电路不会自动断开。
FailureCount	Integer	5	滚动窗口内触发断路器的绝对失败次数。
FailureRatePercent	Integer	50	触发断路器的失败百分比（0-100），在达到 MinCalls 后生效。
SlowCallDurationMs	Integer	3000	成功调用中超过该阈值（毫秒）的调用也将被计为"慢速"调用。
SlowCallRatePercent	Integer	0	触发断路器的慢速调用百分比（0-100）。0 禁用慢速调用检测。

MinCalls	Integer	10	在评估速率阈值之前所需观测到的最少调用次数（避免在小样本上误触发）。

```
// 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

功能说明： 定义用于阈值评估的滚动窗口。窗口被划分为 BucketCount 个子桶，以便旧计数器随时间平滑过期。阈值始终根据有效桶的总和评估。

使用场景： 使用 30 秒滚动窗口和 6 个桶，使瞬时峰值不会触发电路，但持续失败会触发。较长的窗口可以平滑噪声；较短的窗口对中断的反应更快。

属性	类型	默认	描述
RollingWindowSec	Integer	60	滚动窗口的总宽度（秒）。超出此范围的样本将被丢弃。
BucketCount	Integer	10	窗口被划分的子桶数量。每个桶保存成功/失败/慢调用计数器。

```
// 30-second window split into 6 buckets (5 seconds each)
sgcWSCircuitBreaker1->TimeWindow->RollingWindowSec = 30;
sgcWSCircuitBreaker1->TimeWindow->BucketCount = 6;
```

恢复

功能： 控制回路如何从 Open 状态恢复。在 CooldownSec 时间过后，断路器转换到 HalfOpen 状态，并允许 HalfOpenTrialCalls 次探测调用通过。如果探测成功，断路器关闭；如果失败，断路器重新打开，冷却计时器重新开始。

使用时机： 打开后等待 30 秒，然后允许一次试验性调用通过以测试恢复情况，随后再重新关闭。较长的冷却时间可保护正在挣扎的上游服务免受新一波流量的冲击；较短的冷却时间则可使仅出现短暂故障的服务快速恢复。

属性	类型	默认	描述
CooldownSec	Integer	30	在过渡到 HalfOpen 状态前，在 Open 状态下等待的秒数。
HalfOpenTrialCalls	Integer	1	HalfOpen 状态下决定关闭或重新打开之前允许的试验调用次数。

AutoReset	Boolean	True	当值为 True 时，计数器在状态转换时自动清零。
MaxRetries	Integer	3	供重试辅助程序使用的信息性上限。断路器本身不执行任何自动重试。

```
// 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;
```

回退

功能说明：配置电路断路器处于 Open 状态时提供的备用载荷。当 Fallback.Enabled 为 True 且 Execute / ExecuteWithResult 拒绝调用时，OnFallback 事件触发，响应以 CachedResponse（或 CustomMessage）初始化。处理程序可在响应返回给调用方之前替换该字符串。

使用场景：当 OpenAI 出现故障时，返回已缓存的最近一次有效响应，使用户看到降级但可用的功能（例如之前生成的摘要），而不是错误提示。使用 CustomMessage 可设置静态的"维护模式"JSON 正文。

属性	类型	默认	描述
已启用	Boolean	False	启用回退路径。当为 False 时，被拒绝的调用直接返回而不触发 OnFallback。
CachedResponse	string	"	回路处于 Open 状态时返回的静态负载。优先于 CustomMessage。
CustomMessage	string	"	CachedResponse 为空时使用的回退负载。
UseLastSuccess	Boolean	False	当值为 True 时，为密钥观察到的最后一次成功响应将作为备用内容提供。

```
// 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;
```

分类

功能：将引发的异常分类为"记录为故障"或"已忽略"。只有记录的异常才会推进故障计数器。MatchMode 选择 RecordAsFailure/IgnoreExceptions 中的字符串与异常文本（ClassName: Message）进行比较的方式。

使用场景：将带有 500/502/503/504 的 `EidHttpException` 视为失败，但忽略 400/401/404 的 `EidHttpException`（业务错误不应触发断路器）。将 `RecordAsFailure` 保留为空意味着每个未被忽略的异常都会被记录。

属性	类型	默认	描述
<code>RecordAsFailure</code>	<code>TStringList</code>	空	与异常文本匹配的模式；匹配的异常计为失败。空列表表示记录所有未被忽略的内容。
<code>IgnoreExceptions</code>	<code>TStringList</code>	空	与异常文本匹配的模式；匹配的异常被丢弃（既不计为成功也不计为失败）。
<code>MatchMode</code>	<code>TsgcCircuitBreakerExceptionMatch</code>	<code>cemContains</code>	比较模式： <code>cemExact</code> 、 <code>cemContains</code> 、 <code>cemStartsWith</code> 或 <code>cemWildcard</code> （* / ?）。

```
// 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

功能说明：基于模式的覆盖规则集合。每个 `TsgcCircuitBreakerEndpointItem` 都有自己的 `Pattern`（支持通配符 * 和 ?）、`OverrideThresholds` 块和 `OverrideRecovery` 块。当活跃键（主机名）与某个已启用的条目匹配时，该条目的设置将在调用期间替换顶级 `Thresholds / Recovery`。

使用场景：为每个 API 设置独立的断路器：`api.openai.com` 的阈值设置得宽松一些（它偶尔在负载下变慢），而 `api.stripe.com` 涉及支付关键业务，应更快打开以快速失败。为每个主机添加一个端点条目并设置经过调整的阈值。

属性	类型	默认	描述
已启用	<code>Boolean</code>	<code>False</code>	启用端点覆盖集合。为 <code>False</code> 时不评估任何条目。
条目	<code>TCollection</code>	空	<code>TsgcCircuitBreakerEndpointItem</code> 的集合。第一个启用的匹配项优先。

每个 `TsgcCircuitBreakerEndpointItem` 公开以下内容：

属性	类型	默认	描述

模式	string	"	与密钥（主机名）匹配的通配符模式。支持 * 和 ?。
已启用	Boolean	True	禁用此条目而不删除它。
OverrideThresholds	TsgcCircuitBreakerThresholds	defaults	当模式匹配时应用的阈值。仅当其自身的 Enabled 为 True 时有效。
OverrideRecovery	TsgcCircuitBreakerRecovery	defaults	模式匹配时应用的恢复设置。

```

sgcWSCircuitBreaker1->PerEndpoint->Enabled = true;

// OpenAI - lenient: 10 failures, 60s cooldown
TsgcCircuitBreakerEndpointItem *oOpenAI = (TsgcCircuitBreakerEndpointItem*)
    sgcWSCircuitBreaker1->PerEndpoint->Items->Add();
oOpenAI->Pattern = "api.openai.com";
oOpenAI->Enabled = true;
oOpenAI->OverrideThresholds->Enabled = true;
oOpenAI->OverrideThresholds->FailureCount = 10;
oOpenAI->OverrideThresholds->FailureRatePercent = 70;
oOpenAI->OverrideRecovery->CooldownSec = 60;

// Stripe - payment critical: open fast, recover slow
TsgcCircuitBreakerEndpointItem *oStripe = (TsgcCircuitBreakerEndpointItem*)
    sgcWSCircuitBreaker1->PerEndpoint->Items->Add();
oStripe->Pattern = "api.stripe.com";
oStripe->Enabled = true;
oStripe->OverrideThresholds->Enabled = true;
oStripe->OverrideThresholds->FailureCount = 3;
oStripe->OverrideThresholds->FailureRatePercent = 30;
oStripe->OverrideRecovery->CooldownSec = 120;

```

指标

功能说明：随着调用流经断路器，以原子方式更新的只读计数器。跨所有密钥/主机名聚合。CurrentOpenBreakers 在每次状态转换后刷新；AverageLatencyMs 是 Execute 观测到的持续时间的滚动平均值。

使用场景：实时仪表板，显示每台主机的电路状态、故障率和拒绝计数。通过 TTimer 轮询 Metrics 以为运营人员绘制仪表，或将其发送到 Prometheus / StatsD 以触发告警。

属性	类型	描述
TotalCalls	Int64	已处理的总调用次数（成功次数 + 已记录的失败次数）。
TotalFailures	Int64	所有密钥的失败总次数。
TotalSuccesses	Int64	所有密钥记录的成功调用总数。
TotalRejected	Int64	由于电路处于 Open 状态而被拒绝的总调用次数。
CurrentOpenBreakers	Integer	当前处于 Open 状态的密钥数量。
AverageLatencyMs	Double	围绕 Execute / ExecuteWithResult 调用测量的运行平均延迟（毫秒）。

```
// Poll from a TTimer and update a dashboard
void __fastcall TForm1::MetricsTimer(TObject *Sender)
{
    LabelCalls->Caption = "Calls: " +
        IntToStr(sgcWSCircuitBreaker1->Metrics->TotalCalls) + " (ok " +
        IntToStr(sgcWSCircuitBreaker1->Metrics->TotalSuccesses) + " / fail " +
        IntToStr(sgcWSCircuitBreaker1->Metrics->TotalFailures) + " / rejected " +
        IntToStr(sgcWSCircuitBreaker1->Metrics->TotalRejected) + ")";
    LabelOpen->Caption = "Open breakers: " +
        IntToStr(sgcWSCircuitBreaker1->Metrics->CurrentOpenBreakers);
    LabelLatency->Caption = "Avg latency: " +
        FormatFloat("0.0", sgcWSCircuitBreaker1->Metrics->AverageLatencyMs) + " ms";
}
```

事件

OnStateChange：当电路在状态之间转换时触发。提供密钥、旧状态和新状态（csClosed、csOpen 或 csHalfOpen）。适用于日志记录、告警和更新实时 UI。

OnCallRejected：当由于电路处于 Open 状态或 HalfOpen 试验配额耗尽而拒绝调用时触发。提供键值和原因字符串（例如 Circuit open）。

OnFallback：在返回回退响应之前触发。*aResponse* 参数（var string）从 Fallback.CachedResponse / CustomMessage 初始化，可由处理程序替换。

OnFailureRecorded：每次记录失败时触发。提供键值和异常文本（ClassName: Message）。用于结构化日志记录，或与上游事故追踪系统进行关联。

OnSlowCall：当一次成功调用超过 Thresholds.SlowCallDurationMs 时触发。提供密钥和以毫秒为单位的测量持续时间。

公开方法

方法	描述
Execute(Key, Action)	在给定键下运行 TProc。首先检查 IsCallAllowed，自动记录成功/失败/慢速，如果操作执行则返回 True。
Execute(Action)	使用 DefaultKey 的重载。
ExecuteWithResult(Key, Action, out Result)	（D2009+）运行 TFunc<TObject> 并通过 out 参数返回其结果；与 Execute 的计费方式相同。
RecordSuccess(Key)	记录成功的调用。当试验成功时，推进 HalfOpen → Closed。
RecordSuccess	使用 DefaultKey 的重载。
RecordFailure(Key, Exception)	记录失败信息及可选的异常文本，并触发状态转换评估。
RecordFailure(Exception)	使用 DefaultKey 的重载。
ForceOpen(Key)	手动将电路强制切换到 Open 状态。

ForceClose(Key)	手动强制将断路器切换到关闭状态。
Reset(Key)	清除给定键的计数器和状态。
ResetAll	清除每个被跟踪的回路并重置 Metrics。
GetState(Key)	返回密钥的当前 TsgcCircuitState。
IsCallAllowed(Key)	若当前允许发起新的调用，则返回 True。在每次请求前由 HTTP API 客户端自动调用。
GetFailureRate(Key)	返回在滚动窗口中观察到的当前失败百分比（0.0 - 100.0）。
GetTotalCalls(Key)	返回该密钥在滚动窗口中观察到的调用次数。
GetLastStateChange(Key)	返回最后一次状态转换的 TDateTime。
GetOpenBreakers	返回当前处于 Open 状态的键的 TArray<string>。
SaveStateToFile(FileName)	将每个断路器的状态和计数器持久化到文件中。
LoadStateFromFile(FileName)	从之前保存的文件恢复电路状态。
IsConnectionAllowed(IP)	服务器端自我保护钩子，作为客户端主用途的辅助。当 ServerKey 断路器未处于 Open 状态时返回 True。
IsMessageAllowed(IP, Message)	服务器端自我保护钩子。当 ServerKey 电路未处于 Open 状态时返回 True。
RegisterConnection(IP)	保留用于未来按 IP 统计指标的服务器端钩子（目前为空操作）。
UnregisterConnection(IP)	保留用于未来按 IP 统计指标的服务器端钩子（目前为空操作）。
RecordMessageError(IP, Exception)	服务器端自我保护钩子。记录 ServerKey 上的失败。
RecordMessageSuccess(IP)	服务器端自我保护钩子。在 ServerKey 上记录一次成功。

自动集成

当通过 CircuitBreaker 属性将对象分配给 TsgcHTTPAPI_client 后代时，客户端自动：

- **在每次 Get / Post / Put / Patch / Delete 之前：**从目标 URL 主机名（例如 api.openai.com）派生密钥并调用 IsCallAllowed(host)。如果电路处于 Open 状态，调用将被拒绝，抛出异常 *"Circuit breaker open for <host> - call rejected"*，并触发 OnCallRejected。
- **收到成功的 HTTP 响应时：**调用 RecordSuccess(host) 更新计数器，使 HalfOpen 试验能够在远端恢复时关闭回路。
- **发生异常时：**调用 RecordFailure(host, exception.Message)，应用分类规则决定是否计数或忽略，评估状态转换，然后将异常重新抛给调用方。
- **每个主机一个回路：**对不同主机（例如 api.openai.com、api.anthropic.com、api.stripe.com）的调用跟踪完全独立的回路。

完整示例

```
// Enable the circuit breaker
sgcWSCircuitBreaker1->Enabled = true;

// Thresholds: open after 5 failures, or 50% failure rate on 10+ calls
sgcWSCircuitBreaker1->Thresholds->Enabled = true;
sgcWSCircuitBreaker1->Thresholds->FailureCount = 5;
sgcWSCircuitBreaker1->Thresholds->FailureRatePercent = 50;
sgcWSCircuitBreaker1->Thresholds->SlowCallDurationMs = 3000;
sgcWSCircuitBreaker1->Thresholds->MinCalls = 10;

// Rolling window: 60 seconds / 10 buckets
sgcWSCircuitBreaker1->TimeWindow->RollingWindowSec = 60;
sgcWSCircuitBreaker1->TimeWindow->BucketCount = 10;

// Recovery: wait 30s in Open, then 1 trial call in HalfOpen
sgcWSCircuitBreaker1->Recovery->CooldownSec = 30;
sgcWSCircuitBreaker1->Recovery->HalfOpenTrialCalls = 1;

// Classification: record HTTP 5xx and timeouts, ignore 4xx
sgcWSCircuitBreaker1->Classification->MatchMode = cemContains;
sgcWSCircuitBreaker1->Classification->RecordAsFailure->Add("HTTP/1.1 5");
sgcWSCircuitBreaker1->Classification->RecordAsFailure->Add("timeout");
sgcWSCircuitBreaker1->Classification->IgnoreExceptions->Add("HTTP/1.1 404");

// Fallback: serve a cached payload while the circuit is Open
sgcWSCircuitBreaker1->Fallback->Enabled = true;
sgcWSCircuitBreaker1->Fallback->CustomMessage =
    "{ \"error\": \"service unavailable\" }";

// Per-endpoint overrides
sgcWSCircuitBreaker1->PerEndpoint->Enabled = true;
TsgcCircuitBreakerEndpointItem *oStripe = (TsgcCircuitBreakerEndpointItem*)
    sgcWSCircuitBreaker1->PerEndpoint->Items->Add();
oStripe->Pattern = "api.stripe.com";
oStripe->OverrideThresholds->Enabled = true;
oStripe->OverrideThresholds->FailureCount = 3;
oStripe->OverrideRecovery->CooldownSec = 120;

// Persist circuit state across restarts
sgcWSCircuitBreaker1->LoadStateFromFile("circuit.dat");

// Attach to an HTTP API client (any TsgcHTTAPI_client descendant)
sgcAI_OpenAI1->CircuitBreaker = sgcWSCircuitBreaker1;

// Use the client normally; the breaker intercepts Get / Post / etc.
try
{
    String Response = sgcAI_OpenAI1->Get("https://api.openai.com/v1/models");
}
catch (Exception &E)
{
    if (E.Message.Pos("Circuit breaker open") > 0)
        Log("Circuit open for api.openai.com - using fallback");
    else
        Log("Call failed: " + E.Message);
}

// Handle state changes
void __fastcall TForm1::CircuitStateChange(TObject *Sender,
    const String aKey, TsgcCircuitState aOldState, TsgcCircuitState aNewState)
{
    Log("Circuit " + aKey + ": " + IntToStr((int)aOldState) + " -> " +
        IntToStr((int)aNewState));
}

// Handle fallback responses
void __fastcall TForm1::CircuitFallback(TObject *Sender,
    const String aKey, String &aResponse)
{
    Log("Fallback used for " + aKey);
}
```

线程安全

电路断路器完全线程安全。所有公共方法使用内部临界区和线程安全列表来保护对每个键的状态、滚动窗口和指标计数器的并发访问。单个组件可以由多个 HTTP API 客户端共享，并可从任何线程（事件处理程序、定时器、后台工作线程）访问，无需外部同步。

参考

- 属性 — 组件上有 10 个属性可用。
- 方法 — 17 个公开方法。
- 事件 — 5 个可处理的事件。

TsgcWSAPIKeyManager

TsgcWSAPIKeyManager 为 sgcWebSockets 服务器颁发的 API 密钥提供完整生命周期管理。

简介

TsgcWSAPIKeyManager 为 sgcWebSockets 服务器颁发的 API 密钥提供全生命周期管理。它生成具有可配置人类可读前缀的密码学随机密钥，使用 SHA-256、SHA-512 或 Bcrypt 对其进行静态哈希，通过头部或查询字符串验证传入请求，以可配置的宽限期轮换密钥，撤销和更新密钥，执行基于范围的授权，维护持久审计日志，并支持可选的 IP 允许列表。密钥存储在内存或加密文件中，可在重新启动后恢复。

管理器通过 **APIKeyManager** 属性连接到 TsgcWebSocketHTTPServer 和 TsgcWSServer_HTTPAPI。由于服务器的自动钩子只能看到对端 IP，实际的请求身份验证通过从服务器的 OnConnect 事件调用 **IsRequestAuthorized** 来执行——这提供了对完整 HTTP 头块和 URL/查询字符串的访问，以便可以解析、验证已配置的 **X-API-Key** 头（或查询参数），并在需要时与所需作用域进行匹配。

快速入门

1. 将 **TsgcWSAPIKeyManager** 组件拖放到窗体上。
2. 配置生成（前缀、长度）、哈希（算法、盐值）和验证（头名称、IP 白名单、FailClosed）。详见下面各节的文档。
3. 将管理器分配给服务器组件：

```
sgcWebSocketHTTPServer1.APIKeyManager := sgcWSAPIKeyManager1;
```
4. 将服务器的 OnConnect 与 **IsRequestAuthorized** 关联，使用连接的标头、URL 和 IP 进行调用。当返回 False 时断开客户端连接。
5. 为客户签发第一个密钥，并将明文值仅传递给他们一次，此后存储中只保留哈希形式。

顶级属性

功能说明：整个组件的主开关。当 Enabled 为 False 时，所有公共方法将直接返回"允许"，服务器自动钩子变为空操作，密钥管理器实际上被绕过。

使用场景：在受控维护窗口或暂存部署期间，在不取消组件与服务器关联的情况下暂时禁用密钥强制执行。更改完成后重新启用，不会丢失任何密钥，作用域不变，审计日志在您重新打开时立即恢复运行。

属性	类型	默认	描述
已启用	Boolean	True	主开关。为 False 时，ValidateKey / IsRequestAuthorized 始终返回 True，服务器自动钩子变为空操作。

```
sgcWSAPIKeyManager1->Enabled = true;
```

生成

功能说明：控制 *IssueKey* 构建明文密钥字符串的方式。密钥由 *KeyPrefix* 后跟从 *Charset* 中随机抽取的 *KeyLength* 个字符组成。当 *IncludeChecksum* 为 *True* 时，会附加一个短校验和，以便在哈希查找之前快速拒绝格式错误的密钥。

何时使用：为您的 SaaS 提供品牌密钥，在生产环境使用 *sgc_live_aB3...*，在预发布集群使用 *sgc_test_...*。前缀使支持人员能轻松识别泄露密钥所属的环境，也方便日志清理工具对其进行脱敏处理。

属性	类型	默认	描述
已启用	Boolean	True	当为 <i>False</i> 时， <i>IssueKey</i> 将引发异常而非生成新密钥。
KeyPrefix	string	"	随机正文前添加的人类可读前缀（例如 <i>sgc_live_</i> ）。
KeyLength	Integer	32	密钥主体中的随机字符数量。
字符集	string	A-Za-z0-9	用于生成随机请求体的字符表。为空时默认使用 URL 安全的字母数字字符。
IncludeChecksum	Boolean	False	当值为 <i>True</i> 时，附加一个简短的校验和后缀，以便在不进行哈希查询的情况下拒绝明显格式错误的密钥。

```
// Branded production keys: sgc_live_aB3...
sgcWSAPIKeyManager1->Generation->Enabled = true;
sgcWSAPIKeyManager1->Generation->KeyPrefix = "sgc_live_";
sgcWSAPIKeyManager1->Generation->KeyLength = 32;
sgcWSAPIKeyManager1->Generation->IncludeChecksum = true;
```

存储

功能：控制哈希密钥和审计日志的存储位置。*kstMemory* 将所有内容保存在进程内（关闭时清除）。*kstFile* 持久化到 *FileName*，可选地使用 *EncryptionKey* 加密，并每隔 *AutoSaveSeconds* 秒自动保存。*kstCustom* 允许您实现自己的 *SaveToFile* / *LoadFromFile* 挂钩。

使用时机：在开发期间将密钥保存在内存中以加快迭代。对于生产环境，切换到带有 *EncryptAtRest := True* 的 *kstFile*，以便密钥在服务器重启后仍能保留，且不会以明文形式暴露在磁盘上。

属性	类型	默认	描述
StorageType	TsgcAPIKeyStorageType	kstMemory	kstMemory（易失性）、kstFile（持久化到 <i>FileName</i> ）或 kstCustom（用户提供的 <i>SaveToFile/LoadFromFile</i> ）。

FileName	string	"	当 StorageType 为 kstFile 时，SaveToFile / LoadFromFile 使用的文件路径。
EncryptAtRest	Boolean	False	为 True 时，使用 EncryptionKey 对磁盘上的密钥存储进行加密。
EncryptionKey	string	"	用于加密持久化密钥文件的密钥。当 EncryptAtRest 为 True 时必须提供。
AutoSaveSeconds	Integer	0	当值大于 0 时，管理器每隔 AutoSaveSeconds 秒将密钥存储重新保存到磁盘。

```
// 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");
```

哈希

功能：明文密钥永远不会持久化。IssueKey 使用带有可选 Salt 和 Iterations 计数的 Algorithm 对生成的密钥进行哈希；ValidateKey 重新哈希传入值并对存储的摘要执行恒定时间比较。

使用场景：切勿存储明文。SHA-256 是大多数部署的合理默认选择。对于更高安全性配置文件（FIPS 140-2 Level 3），请使用 SHA-512。当需要刻意进行密钥拉伸时使用 Bcrypt——验证速度较慢，但能大幅增加泄露密钥存储的离线暴力破解难度。

属性	类型	默认	描述
算法	TsgcAPIKeyHashAlgorithm	khaSHA256	用于静态存储的哈希函数：khaSHA256、khaSHA512 或 khaBcrypt。
Salt	string	"	混入每个哈希值的静态盐。更改后将使所有现有密钥失效。
迭代	Integer	1	哈希轮数。以验证延迟为代价增加以实现密钥延伸。

```
// 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;
```

范围

功能：允许的范围字符串目录。当 *Enabled* 为 *True* 时，附加到颁发密钥的每个范围必须存在于此列表中；未知范围将被拒绝。各个目录条目可以在不删除的情况下禁用。*ValidateKey / IsRequestAuthorized* 可以要求每个请求使用特定范围。

使用场景：细粒度权限。某个客户的密钥仅携带 *read:orders*，在写入端点上将被拒绝。合作伙伴集成获得 *read:orders + write:shipments*，但永远不会获得 *admin:**。每个请求提供一个范围提示，管理器一致地执行它。

属性	类型	默认	描述
已启用	Boolean	True	启用作用域强制执行。当为 <i>False</i> 时，作用域参数被忽略。
范围	TCollection	空	定义允许范围的 <i>TsgcAPIKeyScopeItem</i> 条目集合。

每个 *TsgcAPIKeyScopeItem* 公开以下内容：

属性	类型	默认	描述
名称	string	"	作用域标识符（例如 <i>read:orders</i> ）。区分大小写。
描述	string	"	在对象检查器和管理界面中显示的人类可读描述。
已启用	Boolean	True	禁用此作用域而不删除它。已禁用的作用域永远无法满足 <i>HasScope</i> 。

```
sgcWSAPIKeyManager1->Scopes->Enabled = true;

TsgcAPIKeyScopeItem *scopeRead = (TsgcAPIKeyScopeItem*)
    sgcWSAPIKeyManager1->Scopes->Scopes->Add();
scopeRead->Name = "read:orders";
scopeRead->Description = "List and read order history";

TsgcAPIKeyScopeItem *scopeWrite = (TsgcAPIKeyScopeItem*)
    sgcWSAPIKeyManager1->Scopes->Scopes->Add();
scopeWrite->Name = "write:shipments";
scopeWrite->Description = "Create and update shipment records";
```

轮换

功能说明：使密钥轮换透明化。*RotateKey* 为同一所有者和权限范围签发新密钥，并将旧密钥标记为 *kksRotated*。在 *GracePeriodSec* 窗口期内，两个密钥均有效，使客户可以无停机地完成切换。当 *AutoRotateDays* 大于 0 时，超过该阈值的旧密钥将由后台扫描自动轮换；*NotifyBeforeExpirySec* 提前触发 *OnKeyExpired*，以便您向客户发送邮件通知。

适用场景：客户报告其密钥已泄露，但其集成仍在运行。轮换密钥：客户会收到一个新密钥，旧密钥在 24 小时内继续有效，以便客户在其基础设施中完成替换，之后旧密钥将自动失效。

属性	类型	默认	描述

已启用	Boolean	False	启用轮换功能（宽限期和自动轮换扫描）。
GracePeriodSec	Integer	86400	RotateKey 后前一个密钥保持有效的秒数。默认为 24 小时。
AutoRotateDays	Integer	0	当 > 0 时，后台扫描将轮换超过此天数的旧密钥。
NotifyBeforeExpirySec	Integer	604800	密钥实际过期前的这么多秒触发 OnKeyExpired。默认为 7 天。

```
// 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;
```

过期时间

功能说明：控制已签发密钥的老化方式。当调用者传入 0 作为 ExpiresInSec 时，DefaultTTLSec 将应用于 IssueKey。EnforceExpiry 切换是否拒绝已过期的密钥。CheckIntervalSec 控制后台扫描检查过期条目、触发 OnKeyExpired 并更新 Stats 的频率。

使用时机：合作伙伴集成密钥必须在 90 天后过期，合作伙伴必须重新签署协议并获取新密钥。将 DefaultTTLSec 设置为 90×86400 意味着每次 IssueKey 调用都遵守该策略，除非调用方明确覆盖它。

属性	类型	默认	描述
DefaultTTLSec	Integer	0	新颁发密钥的默认生存时间（秒）。0 表示永不过期。
EnforceExpiry	Boolean	True	当为 True 时，ValidateKey 拒绝 ExpiresAt 已过去的密钥。
CheckIntervalSec	Integer	60	后台扫描检查过期键并触发 OnKeyExpired 的频率。

```
// 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

功能：存储每个密钥的速率限制默认值（每个窗口的最大请求数）以及每个密钥。这些值是信息性提示，TsgcWSRateLimiter 实例可以通过 PerAPIKey 范围读取，以应用适合层级的限流——API 密钥管理器本身不限制流量。

使用场景：分层 SaaS。免费层客户配额为每小时 100 次请求，专业层为每小时 10,000 次，企业层为每小时 1,000,000 次。在 API 密钥管理器上配置 DefaultMaxRequests / DefaultWindowSec，并与 TsgcWSRateLimiter.PerAPIKey 配对，以便速率限制器自动读取每个密钥的配额。

属性	类型	默认	描述
已启用	Boolean	False	为每个已颁发的密钥启用按密钥级别的速率限制元数据。
DefaultMaxRequests	Integer	100	分配给新密钥的默认最大每窗口请求数。
DefaultWindowSec	Integer	60	与 DefaultMaxRequests 关联的默认窗口大小（秒）。

```
// 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
```

审计

功能说明：将每个密钥生命周期操作（签发、验证、撤销、轮换、过期、授予/撤销范围、验证失败）记录到内存环形缓冲区；当设置了 LogFile 时，还会将相同条目追加到文件中。IncludeIP 记录请求者 IP；IncludePayload 捕获额外详细信息。RetentionDays 清除较旧的条目；MaxMemoryEntries 限制环形缓冲区的容量。

使用场景：合规要求：每次密钥操作必须记录并保留 12 个月。启用审计，将 LogFile 指向您的 SIEM 投放区，并将 RetentionDays 设为 365。OnAuditEvent 为每条日志条目触发，因此您也可以将其发送到消息总线。

属性	类型	默认	描述
已启用	Boolean	True	启用审计日志。当为 False 时，不记录任何条目，OnAuditEvent 也不会触发。
LogFile	string	"	审计日志追加的可选文件路径。
IncludeIP	Boolean	True	当为 True 时，每条条目均记录请求方的 IP 地址。
IncludePayload	Boolean	False	当为 True 时，每条审计条目的 Details 字段携带额外的上下文。
RetentionDays	Integer	90	超过此天数的条目将由后台清扫任务删除。
MaxMemoryEntries	Integer	10000	内存环形缓冲区中保留的最大条目数。

```
// 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;
```

验证

功能说明：告知管理器客户端如何呈现其密钥以及在什么条件下接受请求。*HeaderName*（默认值 `X-API-Key`）和 *QueryParamName*（默认值 `api_key`）是 `ExtractKeyFromHeaders` / `ExtractKeyFromQuery` 查找的两个位置。*RequireHTTPS* 拒绝明文流量。*IPAllowlist* 将密钥限制为特定的来源 IP。*FailClosed*（默认值 `True`）拒绝管理器无法明确授权的任何请求。

使用场景：仅限企业内部的 API：对每个请求强制使用 HTTPS，并将密钥限定在办公室出口 CIDR 范围内，使即便密钥泄露也无法从企业网络外部滥用。

属性	类型	默认	描述
HeaderName	string	X-API-Key	用于读取传入密钥的 HTTP 请求头名称。
QueryParamName	string	api_key	当标头缺失时用作后备的查询字符串参数。
RequireHTTPS	Boolean	False	当为 <code>True</code> 时，无论密钥是否有效，均拒绝所有非 HTTPS 请求。
IPAllowlist	TStringList	空	当列表不为空时，仅授权其 IP 出现在列表中的请求。
FailClosed	Boolean	True	当为 <code>True</code> 时，管理器无法明确授权的任何请求都将被拒绝。

```
// 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");
```

统计信息

功能说明：只读计数器对象，公开密钥存储的实时统计信息。所有计数器在组件的内部临界区内原子性更新。

使用场景：实时管理员仪表板。从 `TTimer` 轮询 `Stats` 并绘制当前 `ActiveKeys` / `RevokedKeys` / `TotalValidations` / `TotalValidationFailures`，以便操作人员实时查看身份验证压力。

属性	类型	描述
TotalKeys	Int64	此管理实例颁发的密钥总数。

ActiveKeys	Int64	当前处于 kksActive 状态的密钥数量。
RevokedKeys	Int64	当前处于 kksRevoked 状态的密钥数量。
ExpiredKeys	Int64	当前处于 kksExpired 状态的密钥数量。
TotalValidations	Int64	ValidateKey / IsRequestAuthorized 的总调用次数。
TotalValidationFailures	Int64	拒绝该密钥的验证调用总次数。
LastIssuedAt	TDateTime	最近一次 IssueKey 调用的时间戳。

```
// Poll from a TTimer and update a dashboard
void __fastcall TForm1::StatsTimer(TObject *Sender)
{
    LabelActive->Caption = "Active: " + IntToStr(sgcWSAPIKeyManager1->Stats->ActiveKeys);
    LabelRevoked->Caption = "Revoked: " + IntToStr(sgcWSAPIKeyManager1->Stats->RevokedKeys);
    LabelValidations->Caption = "Validations: " + IntToStr(sgcWSAPIKeyManager1->Stats->TotalValidations);
    LabelFailures->Caption = "Failures: " + IntToStr(sgcWSAPIKeyManager1->Stats->TotalValidationFailures);
}
```

事件

OnKeyIssued：当新密钥已颁发时触发。提供所有者、原始明文密钥（最后一次查看机会）和范围数组。请在此处理程序中将明文传递给客户。

OnKeyValidated：每次 ValidateKey 完成时触发。提供密钥、布尔型 *aValid* 标志和机器可读的原因字符串。

OnKeyRevoked：密钥被吊销时触发。提供密钥和吊销原因。

OnKeyRotated：密钥轮换时触发。提供旧密钥和新密钥，以便您通知客户。

OnKeyExpired：当密钥的 ExpiresAt 已过期（或提前 NotifyBeforeExpirySec 秒）时，由后台过期扫描触发。

OnValidation：ValidateKey 执行期间触发的最终决策钩子。将 *Allow* 设为 False 并提供 *Reason* 以拒绝本来有效的密钥（自定义业务规则）。

OnAuditEvent：每条审计日志条目都会触发此事件。提供 TsgcAPIKeyAuditEntry 记录（Timestamp、KeyPrefix、Action、IP、Details），适合转发到外部 SIEM 系统。

公开方法

方法	描述
IssueKey(Owner, Scopes, ExpiresInSec)	生成、哈希并存储新密钥；返回明文（这是唯一可查看明文的机会）。传入 ExpiresInSec = 0 将继承 Expiration.DefaultTTLSec。
ValidateKey(Key, RequiredScope, IP)	验证原始密钥。可选择强制要求指定的作用域，并记录请求者的 IP。成功时返回 True。

RevokeKey(Key, Reason)	将某个密钥标记为已撤销；后续 ValidateKey 调用将返回 False。
RotateKey(OldKey, out NewKey)	为相同所有者和范围颁发新密钥；旧密钥被标记为 kksRotated，并在 GracePeriodSec 期间有效。
RenewKey(Key, NewExpiresInSec)	将密钥的到期时间从现在起延长指定的秒数。
GetKeyInfo(Key)	返回一个 TsgcAPIKeyInfo 记录，包含所有者、范围、状态、时间戳和使用计数。
GetKeysByOwner(Owner)	返回与给定所有者关联的哈希密钥标识符。
ListScopes(Key)	返回附加到密钥的作用域。
IsExpired(Key)	如果密钥已过期，返回 True。
IsRevoked(Key)	如果密钥已撤销，返回 True。
KeyExists(Key)	如果密钥已知（活跃、已吊销或已过期），则返回 True。
GrantScope(Key, Scope)	向现有密钥添加范围。
RevokeScope(Key, Scope)	从现有密钥中移除一个范围。
HasScope(Key, Scope)	如果密钥具有给定的作用域，则返回 True。
GetAuditLog(Key, MaxEntries)	返回审计条目；当 Key 为空时，返回所有密钥的条目。
ClearAuditLog(Key)	清除特定密钥的审计日志，或在为空时清除所有密钥的审计日志。
SaveToFile(FileName)	将密钥和审计日志持久化到给定文件（为空时回退到 Storage.FileName）。
LoadFromFile(FileName)	从文件恢复密钥和审计日志。
ExportKeys(Stream)	键和审计日志的基于流的导出。
ImportKeys(Stream)	基于流的密钥和审计日志导入。
PurgeExpired	删除所有 ExpiresAt 已过期的密钥。
Count	返回当前存储的键数量。
IsRequestAuthorized(Headers, QueryString, IP, RequiredScope)	一次性授权检查：从头部或查询字符串中提取密钥，验证并可选地强制执行 scope。
ExtractKeyFromHeaders(Headers)	从原始 HTTP 标头块中解析 Validation.HeaderName 值。
ExtractKeyFromQuery(QueryString)	从原始查询字符串中解析 Validation.QueryParamName 的值。
IsConnectionAllowed(IP)	服务器钩子：当 FailClosed 为 True 且 IPAllowlist 非空且 IP 不在列表中时返回 False。自动调用。
IsMessageAllowed(IP, Message)	服务器钩子，自动为每条入站消息调用。默认情况下在 IP 层无操作。

RegisterConnection(IP)	跟踪新连接。由服务器自动调用。
UnregisterConnection(IP)	断开连接时自动释放连接的追踪记录。

自动集成

当组件分配给服务器的 APIKeyManager 属性时，组件自动：

- **新连接时**：服务器调用 IsConnectionAllowed(IP)，当 FailClosed 为 True 时强制执行可选的 IPAllowlist（否则为空操作）。调用 RegisterConnection(IP) 跟踪活跃连接。
- **收到消息时**：服务器调用 IsMessageAllowed(IP, Message)，在 IP 级别默认为无操作，除非允许列表拒绝该 IP。这些钩子不检查 HTTP 头，因此不在此处执行 API 密钥身份验证。
- **请求授权（用户必须连接）**：在服务器的 OnConnect 事件中调用 *IsRequestAuthorized(Connection.HeadersRequest.Text, Connection.URL, Connection.IP, RequiredScope)*。这是原始 HTTP 标头可用的地方，因此也是提取、验证和可选范围检查 X-API-Key 标头的地方。当函数返回 False 时断开连接。
- **客户端**：在客户端的 OnHandshake/OnBeforeConnect 事件中设置 X-API-Key 标头（或配置的 HeaderName），以便服务器在 HTTP 升级期间读取它。
- **断开连接时**：调用 UnregisterConnection(IP) 以释放内部跟踪。

完整示例

```
// Enable and configure the API key manager
sgcWSAPIKeyManager1->Enabled = true;

// Branded keys
sgcWSAPIKeyManager1->Generation->KeyPrefix = "sgc_live_";
sgcWSAPIKeyManager1->Generation->KeyLength = 32;

// Persist to an encrypted file that survives restarts
sgcWSAPIKeyManager1->Storage->StorageType = kstFile;
sgcWSAPIKeyManager1->Storage->FileName = "apikeys.dat";
sgcWSAPIKeyManager1->Storage->EncryptAtRest = true;
sgcWSAPIKeyManager1->Storage->EncryptionKey = "master-secret";
sgcWSAPIKeyManager1->Storage->AutoSaveSeconds = 60;

// SHA-256 with a static salt
sgcWSAPIKeyManager1->Hashing->Algorithm = khaSHA256;
sgcWSAPIKeyManager1->Hashing->Salt = "my-salt";

// Scope catalog
sgcWSAPIKeyManager1->Scopes->Enabled = true;
TsgcAPIKeyScopeItem *scopeRead = (TsgcAPIKeyScopeItem*)
    sgcWSAPIKeyManager1->Scopes->Scopes->Add();
scopeRead->Name = "read:orders";
scopeRead->Description = "Read orders";
TsgcAPIKeyScopeItem *scopeWrite = (TsgcAPIKeyScopeItem*)
    sgcWSAPIKeyManager1->Scopes->Scopes->Add();
scopeWrite->Name = "write:orders";
scopeWrite->Description = "Create orders";

// 90-day default TTL and 24-hour rotation grace period
sgcWSAPIKeyManager1->Expiration->DefaultTTLSec = 90 * 86400;
sgcWSAPIKeyManager1->Rotation->Enabled = true;
sgcWSAPIKeyManager1->Rotation->GracePeriodSec = 86400;

// Full audit log, 12-month retention
sgcWSAPIKeyManager1->Audit->Enabled = true;
sgcWSAPIKeyManager1->Audit->LogFile = "apikeys-audit.log";
sgcWSAPIKeyManager1->Audit->IncludeIP = true;
sgcWSAPIKeyManager1->Audit->RetentionDays = 365;

// Validation: header-based, fail closed
sgcWSAPIKeyManager1->Validation->HeaderName = "X-API-Key";
sgcWSAPIKeyManager1->Validation->QueryParamName = "api_key";
sgcWSAPIKeyManager1->Validation->FailClosed = true;
```

```
// Load previously saved keys and attach to server
sgcWSAPIKeyManager1->LoadFromFile("apikey.dat");
sgcWebsocketHTTPServer1->APIKeyManager = sgcWSAPIKeyManager1;
sgcWebsocketHTTPServer1->Active = true;

// Issue a new key for a customer (plaintext returned only here)
DynamicArray<String> scopes;
scopes.Length = 2;
scopes[0] = "read:orders";
scopes[1] = "write:orders";
String LKey = sgcWSAPIKeyManager1->IssueKey("customer-123", scopes, 30 * 86400);
String LNewKey;
ShowMessage("Deliver to customer: " + LKey);

// Authorize every incoming connection in OnConnect
void __fastcall TForm1::ServerConnect(TsgcWSConnection *Connection)
{
    if (!sgcWSAPIKeyManager1->IsRequestAuthorized(
        Connection->HeadersRequest->Text,
        Connection->URL, Connection->IP, "read:orders"))
        Connection->Disconnect();
}

// Rotate a leaked key (old one valid 24h during grace)
sgcWSAPIKeyManager1->RotateKey(LKey, LNewKey);

// Revoke a key on subscription cancellation
sgcWSAPIKeyManager1->RevokeKey(LNewKey, "customer cancelled subscription");
```

线程安全

API 密钥管理器是完全线程安全的。所有公共方法使用内部临界区和线程安全列表来保护对密钥存储、范围索引、所有者索引和审计日志的并发访问。单个组件可以被多个服务器实例共享，并可从任意线程（服务器事件处理程序、计时器线程、后台工作线程）访问，无需外部同步。

参考

- 属性 — 组件上有 10 个可用属性。
- Methods — 26 个公有方法。
- 事件 — 7 个可处理的事件。

TsgcWebSocketLoadBalancerServer

TsgcWebSocketLoadBalancerServer 组件。

简介

组件 **TsgcWebSocketLoadBalancerServer** 允许您对 **WebSocket** 和 **HTTP** 协议进行负载均衡。对于 **WebSocket** 协议，它将消息分发到一组服务器，并使用随机序列或最少连接算法分配客户端连接。

负载均衡服务器继承了 **TsgcWebSocketHTTPServer** 的所有方法和属性。

负载均衡器配置

负载均衡器服务器是 **TsgcWebSocketHTTPServer** 的后代，因此请阅读 **TsgcWebSocketHTTPServer** 的文档以了解如何配置它。

此外，负载均衡器具有 **LoadBalancer** 属性，该属性包含以下子属性：

- **LoadBalancing**：在此配置如何分发连接
 - **lbRandom**：（默认）每次新客户端请求连接时，返回一个随机服务器。
 - **lbConnections**：每次新客户端请求连接时，返回已连接客户端最少的服务器。
- 协议：配置后用哪些协议
 - **WebSocket**：若为 true，WebSocket 连接将由负载均衡服务器处理。
 - **HTTP**：若为 true，HTTP 连接将由负载均衡服务器处理。

备用服务器配置

备份服务器（负载均衡器后端服务器）可以是 **TsgcWebSocketServer**、**TsgcWebSocketHTTPServer** 或 **Datasnap 服务器**。

这些服务器有一个名为 **LoadBalancer** 的属性，您可以在其中配置负载均衡服务器与备用服务器之间的连接。

- **Enabled**：若要将其用作备份服务器，请设置为 true。
- **Host**：LoadBalancer 所在的主机。
- **Port**：LoadBalancer 的监听端口。
- **Guid**：标识此服务器的唯一 ID。
- **Bindings**：连接将被转发到的公共地址。示例：若备用 WebSocket 服务器正在端口 8000 上监听，且 IP 地址为 1.1.1.1，则使用如下格式：ws://1.1.1.1:8000;
- **AutoRegisterBindings**：若启用，负载均衡服务器将使用备份服务器的 Bindings 属性来配置公共绑定。
- **AutoRestart**：以秒为单位，如果大于零，备份服务器的负载均衡器客户端将启用内部看门狗，每 x 秒检查一次连接是否存活，如果连接已关闭，将尝试重新连接。

参考

- 属性 — 组件上提供 18 个属性。
- 方法 — 9 个公开方法。
- 事件 — 22 个可处理的事件。

TsgcWebSocketProxyServer

TsgcWebSocketProxyServer 实现了一个 WebSocket 服务器组件，监听客户端 WebSocket 连接并将数据连接转发到普通 TCP/IP 服务器。

简介

TsgcWebSocketProxyServer 实现了一个 WebSocket 服务器组件，它监听客户端 WebSocket 连接并将数据连接转发到普通 TCP/IP 服务器。这对浏览器连接尤为有用，因为它允许浏览器虚拟连接到任意服务器。

参考

- 属性——组件上共有 19 个可用属性。
- 方法 — 7 个公共方法。
- 事件 — 22 个可处理的事件。

TsgclWWebSocketClient

TsgclWWebSocketClient 实现了一个 *IntraWeb WebSocket* 组件，可连接到 *WebSocket* 服务器。

简介

TsgclWWebSocketClient 实现了一个 *IntraWeb WebSocket* 组件，可以连接到 *WebSocket* 服务器。请按照以下步骤配置此组件：

1. 在窗体上拖放 *TsgclWWebSocketClient* 组件
2. 设置 *Host* 和 *Port*（默认为 80）以连接到可用的 *WebSocket* 服务器。您可以设置 *URL* 属性，*Host*、*Port*、*Parameters* 等将从 *URL* 自动更新。例如：*wss://127.0.0.1:8080/ws/* 将得到：

```
oClient = new TsgclWWebSocketClient();  
oClient->Host = "127.0.0.1";  
oClient->Port = 80;  
oClient->TLS = true;  
oClient->Options->Parameters = "/ws/";
```

3. 你可以选择是否需要 *TLS*（安全连接），默认情况下它未启用。
4. 设置允许的传输方式。

WebSockets：将使用标准 *WebSocket* 实现。

Emulation：如果浏览器不支持 *WebSockets*，那么它将使用循环 *AJAX* 回调连接

6. 创建异步过程并将属性 *Active* 设置为 *True*

Methods

Open：打开 *WebSocket* 连接。

Close：关闭 *WebSocket* 连接。

WriteData：sends a message to *WebSocket Server*。

Properties

Connected：是一个只读变量，如果连接处于活跃状态则返回 *True*，否则返回 *False*。

JSOpen：您可以在此处包含在连接打开时在客户端执行的 JavaScript 代码。

JSClose：此处可包含当连接关闭时在客户端执行的 JavaScript 代码。

JSMessage：在这里你可以包含 当客户端从服务器接收到消息时在客户端运行的 JavaScript 代码。你可以使用 JavaScript 变量 "text" 获取消息字符串。

JSError：此处可包含客户端 JavaScript 代码，在发生错误时执行。可通过 Javascript 变量 "text" 获取错误消息。

参考

- 属性 — 组件上有 11 个可用属性。
- 方法 — 4 个公共方法。
- 事件 — 5 个可处理的事件。

TsgcWSConnection

TsgcWSConnection 是客户端 WebSocket 连接的包装器。您可以在服务器或客户端事件中访问此对象。

方法

WriteData：向客户端发送消息。

关闭：向对端发送关闭消息。可选指定"CloseCode"。默认情况下，发送的值为正常关闭代码。如果您发送负数关闭代码，则不会发送关闭原因。

Disconnect：从服务器端关闭客户端连接。可以选择指定"CloseCode"。

Ping：向客户端发送 ping。

AddTCPEndOfFrame：如果连接为普通 TCP，允许您设置定义消息结尾的字节。消息将被缓冲直到完整接收。

Subscribed：返回连接是否已订阅自定义频道。

Subscribe：将此连接订阅到频道。之后，您可以从服务器组件向订阅了该频道的所有连接广播消息。

UnSubscribe：取消订阅此连接的某个频道。

属性

Protocol：返回此连接使用的子协议。

IP：返回对等方 IP 地址。

Port：返回对端端口。

LocalIP：返回主机 IP 地址。

LocalPort：返回主机端口。

URL：返回客户端请求的 URL。

Guid：返回连接 ID。

HeadersRequest：返回请求中收到的标头列表。

HeadersResponse：返回作为响应发送的标头列表。

RecBytes：已接收的字节数。

SendBytes：已发送的字节数。

Transport：返回连接的传输类型：

trpRFC6455：普通的 WebSocket 连接。

trpHixie76：使用草案 WebSocket 规范的 WebSocket 连接。

trpFlash：使用 Flash 作为回退的 WebSocket 连接。

trpSSE：服务器发送事件（SSE）连接。

trpTCP：普通 TCP 连接。

TCPEndOfFrameScanBuffer：允许定义用于查找消息结尾的方法（当使用 trpTCP 作为传输时）。

eofScanNone：每次有新数据包到达时，都会调用 OnBinary 事件。

eofScanLatestBytes：如果最新字节等于通过 AddTCPEndOfFrame 方法添加的字节，则调用 OnBinary 消息；否则缓冲此数据包

eofScanAllBytes：在整个数据包中搜索与 AddTCPEndOfFrame 方法添加的字节相等的字节。若找到，则调用 OnBinary 事件，否则对该数据包进行缓冲。

Data：用户会话数据对象，您可以在此处传入一个对象，并在每次需要时访问它，例如：您可以传入数据库连接、用户会话属性.....

协议

使用 WebSocket，您可以实现子协议，从而创建自定义通信。**例如**，您可以在 WebSocket 协议之上实现一个子协议，使用 JSON 消息与自定义应用程序通信，也可以使用 XML 消息实现另一个子协议。

当服务器端打开连接时，将验证客户端发送的子协议是否受服务器支持；若不支持，则关闭连接。服务器可以实现多个子协议，但单个连接上只能使用一个。

子协议对于创建自定义应用程序以及确保所有客户端支持相同的通信接口非常有用。

尽管协议名称可以任意指定，但建议使用唯一名称，例如"dataset.esegce.com"

使用 sgcWebSockets 包，您可以构建自己的协议，也可以使用内置的子协议：

1. **协议 MQTT**：MQTT 是一种客户端/服务器发布/订阅消息传输协议。它轻量、开放、简单，且易于实现。
2. **Protocol AppRTC**：这是由 Google 和 Mozilla 开发的 WebRTC 演示应用程序，使两个浏览器能够使用 WebRTC API 互相"通话"。
3. **Protocol WebRTC**：旨在为 Web 带来实时通信（RTC）能力的开源项目。
4. **Protocol Files**：使用二进制消息实现，提供文件发送支持：数据包大小、授权、QoS、消息确认等。
5. **SGC 协议**：使用 JSON-RPC 2.0 消息实现，提供以下模式：RPC、发布/订阅、事务性消息、消息确认等更多功能。
6. **Protocol Dataset**：继承自默认协议，可将数据集变更（新记录、保存记录或删除记录）从服务器发送至客户端。
7. **Protocol Presence**：允许您了解谁订阅了某个频道，例如：聊天室、文档协作者、查看同一网页的人、游戏中的竞争对手.....
8. **协议 WAMP 1.0**：开放的 WebSocket 子协议，提供 RPC 和 PubSub 两种异步消息模式。
9. **Protocol WAMP 2.0**：开放的 WebSocket 子协议，提供两种异步消息传递模式：RPC 和 PubSub。
10. **协议 STOMP**：STOMP 是简单（或流式）面向文本的消息协议。STOMP 提供了一种可互操作的线路格式，使 STOMP 客户端能够与任何 STOMP 消息代理通信，从而在多种语言、平台和代理之间提供简便、广泛的消息互操作性。
 - 10.1 **STOMP for RabbitMQ**：用于 RabbitMQ Broker 的客户端。
 - 10.2 **ActiveMQ 的 STOMP**：ActiveMQ Broker 的客户端。
11. **Protocol AMQP**：高级消息队列协议（AMQP 0.9.1）是作为开放标准协议创建的，允许系统之间进行消息互通，无论使用何种消息代理厂商或平台。
12. **Protocol AMQP1**：高级消息队列协议（AMQP 1.0.0）作为开放标准协议创建，允许系统之间的消息互操作，不受消息代理供应商或平台限制。
13. **Protocol E2EE**：通过 WebSocket 连接发送的消息在应用层进行端到端加密，因此即使 WebSocket 服务器中继数据，也只有通信中的客户端才能读取。

如果您需要**通过单个连接使用多个协议**（例如：您可能需要使用**默认协议**处理远程过程调用，同时使用**Dataset 协议**处理数据库连接），可以为每个协议组件分配一个"Broker"，所有消息将通过该中介协议进行交换（可参考"Tickets Demo"获取简单示例）。

协议可在**运行时**注册，只需调用方法 **RegisterProtocol**，并将协议组件作为参数传入。

JavaScript 参考

您可以在此处获取有关 sgcWebSockets 中使用的通用 JavaScript 库的[更多信息](#)。

协议 Javascript

默认情况下，sgcWebSockets 使用 **sgcWebSocket.js** 文件。

以下是可用的方法，您可以根据需要替换 `{%host%}` 和 `{%port%}` 变量，例如：如果您已将 sgcWebSocket 服务器配置为在 `www.example.com` 网站的 80 端口上监听，则需要将对 `sgcWebSocket.js` 文件的访问配置为：

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
```

打开连接

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
</script>
```

sgcWebSocket 有 3 个参数，只有第一个是必填的：

```
sgcWebSocket(url, protocol, transport)
```

- **URL**：WebSocket 服务器地址，普通 WebSocket 连接使用 "ws:"，安全 WebSocket 连接使用 "wss:"。

```
sgcWebSocket('ws://127.0.0.1')
```

```
sgcWebSocket('wss://127.0.0.1')
```

- **协议**：如果服务器接受一个或多个协议，您可以定义希望使用的协议。

```
sgcWebSocket('ws://127.0.0.1', 'esegece.com')
```

- **Transport**：默认情况下，首先尝试使用 WebSocket 连接，如果浏览器不支持，则尝试使用 Server Sent Events 作为传输。

如果已实现，请使用 WebSocket；否则使用 Server Sent Events：

```
sgcWebSocket('ws://127.0.0.1')
```

仅使用 WebSocket 作为传输：

```
sgcWebSocket('ws://127.0.0.1', '', ['websocket'])
```

仅使用服务器发送事件 (SSE) 作为传输：

```
sgcWebSocket('ws://127.0.0.1', '', ['sse'])
```

使用身份验证打开连接

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket({"host":"ws://{%host%}:{%port%}", "user":"admin", "password":"1234"});
</script>
```

发送消息

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
  socket.send('Hello sgcWebSockets!');
</script>
```

收到消息时显示提醒

```
<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>
```

收到二进制消息

```
<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>
```

收到二进制（标头 + 图像）消息

```
<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>
```

在 OnConnect、OnDisconnect 和 OnError 事件时显示警告

```
<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 Close!');
  }
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error!');
  }
</script>
```

```
};
socket.on('close', function(event)
{
    alert('sgcWebSocket Closed!');
});
socket.on('error', function(event)
{
    alert('sgcWebSocket Error: ' + event.message);
});
</script>
```

关闭连接

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
    socket.close();
</script>
```

获取连接状态

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
    socket.state();
</script>
```

协议 MQTT

MQTT 是一种基于客户端-服务器的发布/订阅消息传输协议。它轻量、开放、简单，设计上易于实现。这些特性使其非常适合在受限环境中使用，例如 M2M 和 IoT 场景。

该协议 运行在 TCP/IP 上，或运行在提供有序、无损、双向连接的其他网络协议上。其功能包括：

- 使用发布/订阅消息模式，实现一对多消息分发和应用程序解耦。
- 与载荷内容无关的消息传输。
- 消息投递的三种服务质量等级：
 - "最多一次"，消息按操作环境的最佳努力传送，可能发生消息丢失。此级别例如可用于环境传感器数据，即使丢失单条读数也无妨，因为下一条很快就会发布。
 - "至少一次"，消息保证到达，但可能出现重复。
 - "精确一次"，即消息被保证恰好传送一次。例如，此级别可用于计费系统，因为重复或丢失的消息可能导致错误收费。
- 较小的传输开销，协议交换最小化以减少网络流量。
- 一种在发生异常断开连接时通知相关方的机制。

功能特性

- 支持 MQTT 3.1.1 和 5.0 版本。
- 发布/订阅消息模式，用于实现一对多消息分发和应用程序解耦。
- 确认已发送的消息。
- 实现QoS（服务质量）用于消息传递（所有级别：至多一次、至少一次和恰好一次）
- 遗嘱消息。
- 安全连接。
- HeartBeat 和 Watchdog。
- 身份验证到服务器。

组件

TsgcWSPClient_MQTT：MQTT 客户端组件。

最常见的用途

- 连接
 - [客户端 MQTT 连接](#)
 - [连接 Mosquitto MQTT 服务器](#)
 - [客户端 MQTT 会话](#)
 - [Client MQTT Version](#)
- 发布和订阅
 - [MQTT 发布订阅](#)
 - [MQTT 主题](#)
 - [MQTT 订阅](#)
 - [MQTT 发布消息](#)
 - [MQTT 接收消息](#)
 - [MQTT 发布并等待响应](#)
- 其他
 - [MQTT 清除保留消息](#)

TsgcWSPClient_MQTT

MQTT 组件提供轻量、功能完整的 MQTT 客户端实现，支持版本 3。

简介

MQTT 组件提供轻量级、功能完整的 MQTT 客户端实现，支持 3.1.1 和 5.0 版本。该组件支持通过标准 TCP 和 WebSocket 进行明文和安全连接。

连接到 MQTT 服务器非常简单，只需将此组件拖放到表单上，并使用 Client 属性选择一个 [TsgcWebSocketClient](#) 组件。在 TsgcWebSocketClient 中设置主机和端口，然后将 Active := True 即可连接。

MQTT v5.0 与 v3.1.1 不向后兼容，引入了过多新内容，现有实现需要重新审视。

根据规范，MQTT v5.0 在保留大部分核心内容的同时，为 MQTT 增加了大量新特性。

- Clean Session 标志的功能被分为 2 个属性，以允许对会话状态数据进行更精细的控制：CleanStart 参数和新增的 SessionExpInterval。
- 服务器断开连接：允许服务器发送 DISCONNECT 消息，以指示连接关闭的原因。
- 所有响应数据包（CONNACK、PUBACK、PUBREC、PUBREL、PUBCOMP、SUBACK、UNSUBACK、DISCONNECT）现在均包含描述操作成功或失败原因的原因码和原因字符串。
- 增强身份验证：提供一种机制，用于启用包括双向身份验证在内的挑战/响应式身份验证。如果客户端和服务端都支持，则允许使用 SASL 样式的身份验证，并包括客户端在连接内重新进行身份验证的能力。
- 请求/响应模式通过添加 ResponseTopic 而正式化。
- 共享订阅：添加共享订阅支持，允许对一个订阅进行负载均衡消费。
- 主题别名可由客户端和服务端发送，通过较短的数字标识符引用主题过滤器，以节省带宽。
- 服务端可以在 ConnectionProperties 中告知其支持的功能。
- 服务端引用：允许服务端在 CONNACK 或 DISCONNECT 时指定要使用的备用服务端。可用作重定向或设备预配。
- 更多：消息过期、接收最大值和最大数据包大小，以及 Will Delay 间隔，均受支持。

参考

- 属性 — 组件上共有 10 个可用属性。
- 方法 — 9 个公共方法。
- 事件 — 共 13 个可处理的事件。

TsgcWSPClient_MQTT | 客户端 MQTT 连接

要连接到 MQTT 服务器，您必须首先创建一个 [TsgcWebSocketClient](#) 和一个 [TsgcWSPClient_MQTT](#)。然后您必须将 MQTT 组件附加到 WebSocket 客户端。

基本用法

使用 WebSocket 协议连接到 Mosquitto MQTT 服务器。连接后订阅主题："topic1"。

```
oClient = new TsgcWebSocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8080;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;

void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const int ReasonCode,
    const string ReasonName, const TsgcWSMQTTCONNACKProperties *ConnectProperties);
{
    oMQTT->Subscribe("topic1");
}
```

客户端标识符

MQTT 需要客户端标识符来标识客户端连接。组件会自动设置一个随机值，但如有需要，您也可以设置自己的客户端标识符。为此，只需处理 **OnBeforeConnect** 事件并在 `aClientIdentifier` 参数中设置您的值。

```
void OnMQTTBeforeConnect(TsgcWSConnection *Connection, ref bool aCleanSession,
    ref string aClientIdentifier)
{
    aClientIdentifier = "your client id";
}
```

身份验证

某些服务器需要用户名和密码才能授权 MQTT 连接。在连接服务器之前，使用 **Authentication** 属性设置用户名和密码的值。

```
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Authentication->Enabled = true;
oMQTT->Authentication->UserName = "your user";
oMQTT->Authentication->Password = "your password";
```

TsgcWSPClient_MQTT | 连接 MQTT Mosquitto

使用以下示例配置连接到 Mosquitto MQTT 服务器。

MOSQUITTO MQTT WebSockets

```
oClient = new TsgcWebsocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8080;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;
```

MOSQUITTO MQTT WebSockets TLS

```
oClient = new TsgcWebsocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8081;
oClient->TLS = true;
oClient->TLSOptions->Version = tls1_2;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;
```

MOSQUITTO MQTT Plain TCP

```
oClient = new TsgcWebsocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 1883;
oClient->Specifications->RFC6455 = false;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;
```

MOSQUITTO MQTT Plain TCP TLS

```
oClient = new TsgcWebsocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8083;
oClient->Specifications->RFC6455 = false;
oClient->TLS = true;
oClient->TLSOptions->Version = tls1_2;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;
```

TsgcWSPClient_MQTT | 客户端 MQTT 会话

Clean Start

OnMQTTBeforeConnect 事件中有一个名为 **aCleanSession** 的参数。若此参数值为 **True**，表示客户端希望开始新会话，因此若服务器存储了任何会话，则必须丢弃。此时，当 **OnMQTTConnect** 事件触发时，**aSession** 参数将为 **false**。若此参数值为 **False**，且存在与此客户端标识符关联的会话，则服务器必须从现有会话的状态恢复与客户端的通信。

因此，如果客户端发生意外断线，而您希望恢复断线前的会话，请在 **OnMQTTBeforeConnect** 中将 **aCleanSession = True** 和 **aClientIdentifier = 会话的客户端 ID**。

会话

连接成功后，检查 **OnMQTTConnect** 事件中 **Session** 参数的值。

Session = true，表示会话已恢复。

Session = false，表示这是一个新会话。

```
void OnMQTTBeforeConnect(TsgcWSConnection *Connection, ref bool aCleanSession,
    ref string aClientIdentifier)
{
    aCleanSession = false;
    aClientIdentifier = "previous client id";
}

void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const int ReasonCode,
    const string ReasonName, const TsgcWSMQTTCONNACKProperties *ConnectProperties);
{
    if (Session == true)
    {
        WriteLn("Session resumed");
    }
    else
    {
        WriteLn("New Session");
    }
}
```

TsgcWSPClient_MQTT | 客户端 MQTT 版本

目前，MQTT 客户端支持以下规范：

- **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>

您可以使用 MQTTVersion 属性选择 MQTT 客户端组件将使用的版本。

MQTT 3.1.1：TsgcWSPClient_MQTT.Version = mqtt311

MQTT 5.0：sgcWSPClient_MQTT.Version = mqtt5

TsgcWSPClient_MQTT | MQTT 发布订阅

发布/订阅模式（也称为 pub/sub）提供了传统客户端-服务器架构的替代方案。在客户端-服务器模型中，客户端直接与端点通信。pub/sub 模型将发送消息的客户端（发布者）与接收消息的客户端（订阅者）解耦。发布者和订阅者从不直接联系彼此，实际上它们甚至不知道对方的存在。它们之间的连接由第三方组件（代理）处理。代理的任务是过滤所有传入消息并将其正确分发给订阅者。

使用 **TsgcWSPClient_MQTT**，您可以发布消息并订阅主题。

订阅主题

在连接成功后订阅主题"topic1"。

```
oClient = new TsgcWebSocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8080;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;

void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const int ReasonCode,
    const string ReasonName, const TsgcWSMQTTCONNACKProperties *ConnectProperties);
{
    oMQTT->Subscribe("topic1");
}
```

发布消息

向"topic1"的所有订阅者发布消息

```
oClient = new TsgcWebSocketClient();
oClient->Host = "test.mosquitto.org";
oClient->Port = 8080;
oMQTT = new TsgcWSPClient_MQTT();
oMQTT->Client = oClient;
oClient->Active = true;

void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const int ReasonCode,
    const string ReasonName, const TsgcWSMQTTCONNACKProperties *ConnectProperties);
{
    oMQTT->Publish("topic1", "Hello Subscribers topic1");
}
```

TsgcWSPClient_MQTT | MQTT 主题

主题

在 MQTT 中，"主题"是指代理用于过滤每个已连接客户端消息的 UTF-8 字符串。主题由一个或多个主题层级组成，每个主题层级用正斜杠（主题层级分隔符）分隔。

```
myHome / groundfloor / livingroom / temperature
```

与消息队列相比，MQTT 主题非常轻量。客户端无需在发布或订阅之前创建所需的主题。broker 在无需任何预先初始化的情况下接受每个有效主题。请注意，每个主题至少须包含 1 个字符，且主题字符串允许包含空格。主题区分大小写。

WildCards

当客户端订阅某个主题时，它可以订阅已发布消息的确切主题，也可以使用通配符同时订阅多个主题。通配符只能用于订阅主题，不能用于发布消息。通配符有两种类型：单级和多级。

单级：+

顾名思义，单级通配符替换一个主题级别。加号表示主题中的单级通配符。

```
myHome / groundfloor / + / temperature
```

任何主题与具有单级通配符的主题匹配，前提是它包含一个任意字符串代替通配符。例如，订阅 `_myhome/groundfloor+/temperature` 可能产生以下结果：

```
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
```

多级：#

多级通配符覆盖多个主题级别。井号表示主题中的多级通配符。要让 broker 确定哪些主题匹配，多级通配符必须作为最后一个字符放置在主题中，并在其前面加上正斜杠。

```
myHome / groundfloor / #
```

```
YES => myHome / groundfloor / livingroom / temperature
YES => myHome / groundfloor / kitchen / temperature
YES => myHome / groundfloor / kitchen / brightness
NO => myHome / firstfloor / kitchen / temperature
```

当客户端使用多级通配符订阅主题时，它将接收以通配符字符之前的模式开头的所有主题消息，无论主题有多长或多深。如果仅指定多级通配符作为主题（`_#`），您将接收发送到 MQTT broker 的所有消息。

TsgcWSPClient_MQTT | MQTT 订阅

您可以使用 TsgcWSPClient_MQTT 的 Subscribe 方法订阅主题。该方法具有以下参数：

Topic：要订阅的主题名称。

QoS：三个 QoS 级别之一（并非所有 broker 都支持全部 3 个级别）。若未指定，则使用 mtqsAtMostOnce。详细了解 [QoS 级别](#)。

SubscribeProperties：若为 MQTT 5.0，则为关于订阅的附加属性。

订阅 QoS = 至少一次

```
MQTT->Subscribe("topic1", mtqsAtLeastOnce);
```

订阅 MQTT 5.0

```
oProperties = new TsgcWSMQTTSubscribe_Properties();  
oProperties->SubscriptionIdentifier = 1234;  
oProperties->UserProperties->Add("name=value");  
  
MQTT->Subscribe("topic1", mtqsAtMostOnce, oProperties);
```

TsgcWSPClient_MQTT | MQTT 发布消息

您可以使用 **Publish** 方法向某个主题的所有订阅者发布消息，该方法具有以下参数：

Topic：消息将被发布到的主题名称。

Text：消息的文本内容。

QoS：三个 QoS 级别之一（并非所有 broker 都支持全部 3 个级别）。若未指定，则使用 `mtqsAtMostOnce`。详细了解 [QoS 级别](#)。

Retain：若为 `True`，此消息将被保留。每当新客户端订阅此主题时，该消息将发送给该客户端。

PublishProperties：如果使用 MQTT 5.0，这些是消息的属性。

发布简单消息

```
MQTT->Publish("topic1", "Hello Subscribers topic1");
```

发布 QoS = 至少一次

```
MQTT->Publish("topic1", "Hello Subscribers topic1", mtqsAtLeastOnce);
```

发布保留消息

```
MQTT->Publish("topic1", "Hello Subscribers topic1", mtqsAtMostOnce, true);
```


TsgcWSPClient_MQTT | MQTT 接收消息

服务器发送的消息在 **OnMQTTPublish** 事件中接收。该事件包含以下参数：

Topic：与此消息关联的主题名称。

Text：消息的文本内容。

PublishProperties：如果是 MQTT 5.0，这些是已发布消息的属性。

读取已发布的消息

```
void OnMQTTPublish(TsgcWSConnection *Connection, string aTopic, string aText,
    TsgcWSMQTTPublishProperties *PublishProperties)
{
    WriteLine("Topic: " + aTopic + ". Message: " + aText);
}
```

TsgcWSPClient_MQTT | MQTT 接收消息（扩展）

OnMQTTPublishEx 事件通过 **TsgcWSMQTTPublishData** 对象以多种格式提供已发布的消息有效负载。此事件具有以下参数：

Topic：与此消息关联的主题名称。

Data：包含已发布消息的有效载荷。它具有以下属性：

- **Value**：有效载荷（字符串形式）。
- **Bytes**：以 TBytes 形式表示的原始载荷。
- **Stream**：以 TMemoryStream 形式表示的原始载荷。

PublishProperties：如果是 MQTT 5.0，这些是已发布消息的属性。

读取已发布的消息（扩展）

```
void OnMQTTPublishEx(TsgcWSConnection *Connection, string aTopic,
    TsgcWSMQTTPublishData *aData, TsgcWSMQTTPublishProperties *PublishProperties)
{
    // read as string
    WriteLine("Topic: " + aTopic + ". Message: " + aData->Value);
    // read as bytes
    WriteLine("Bytes Length: " + IntToStr(aData->Bytes.Length));
    // read as stream
    WriteLine("Stream Size: " + IntToStr(aData->Stream->Size));
}
```

TsgcWSPClient_MQTT | 发布并等待响应

MQTT 客户端支持使用某些类型的 QoS 级别，每个级别以不同的方式工作，以确保消息按预期处理。

共有以下 QoS 级别：

- **mtqsAtMostOnce**：（默认情况下）消息按照底层 TCP/IP 网络的最大努力传送。协议中不定义响应和重试语义。消息到达服务器的次数为一次或零次。
- **mtqsAtLeastOnce**：服务器收到消息由 ACKNOWLEDGMENT 消息确认。如果通信链路或发送设备发生已识别的故障，或在指定时间内未收到确认消息，发送方将重新发送消息。消息至少到达服务器一次。QoS 级别 1 的消息在消息中有 ID 参数。
- **mtqsExactlyOnce**：消息确保恰好送达一次。例如，此级别可用于计费系统，因为重复或丢失的消息可能导致错误收费。如果通信链路或发送设备发生故障，或在指定时间内未收到确认消息，发送方将重新发送消息。

您可以处理 OnPubAck 或 OnPubComp 事件，以了解消息是否已被服务器处理；也可以使用方法 **PublishAndWait** 来确认消息是否已被服务器处理。

PublishAndWait 的使用方式与普通 Publish 方法相同，但现在有一个名为 Timeout 的新参数，如果在一定时间内没有收到服务器的响应，该方法将返回 false。默认情况下，此值为 10 秒。

```
if mqtt->PublishAndWait("topic", "text")
{
    ShowMessage("Message processed")
}
else
{
    ShowMessage("Message error");
}
```

TsgcWSPClient_MQTT | MQTT 清除保留消息

默认情况下，每个 MQTT 主题都可以有一条保留消息。清理保留消息的标准 MQTT 机制是向主题发送一条空负载的保留消息，这将删除该保留消息。

```
MQTT->Publish("topic1", "", mqttAtMostOnce, true);
```

协议 AMQP 0.9.1

高级消息队列协议（AMQP）是一种面向消息中间件的开放标准应用层协议。AMQP 的定义特性包括：消息导向、队列、路由（包括点对点和发布订阅）、可靠性和安全性。

AMQP 是一种二进制应用层协议，旨在高效支持各种消息传递应用程序和通信模式。它提供流量控制的、面向消息的通信，并保证消息投递：至多一次（每条消息投递一次或从不投递）、至少一次（每条消息确保投递，但可能多次投递）和恰好一次（消息始终确保到达且仅到达一次），以及基于 SASL 和/或 TLS 的身份验证和/或加密。它假定底层有可靠的传输层协议，例如传输控制协议（TCP）。

功能特性

在客户端与代理之间需要高质量、安全消息传递的任何场景下均可使用 AMQP。

AMQP 提供以下功能：

- 监控和共享更新。
- 确保服务器对请求快速响应，并将耗时任务转交后续处理。
- 将消息分发给多个收件人。
- 为进一步数据检索连接离线客户端。
- 提高应用程序的可靠性和稳定运行。
- 消息投递的可靠性。
- 高速消息投递。
- 消息接受。

组件

TsgcWSPClient_AMQP：实现 **AMQP 0.9.1** 协议的客户端组件。

最常见的用途

- 连接
 - [Client AMQP Connect](#)
 - [Client AMQP Disconnect](#)
- 命令
 - [AMQP 频道](#)
 - [AMQP 交换机](#)
 - [AMQP 队列](#)
 - [AMQP 发布消息](#)
 - [AMQP 消费消息](#)（异步）
 - [AMQP 获取消息](#)（同步）
 - [AMQP QoS](#)
 - [AMQP 事务](#)

TsgcWSPClient_AMQP

AMQP 0.

简介

TsgcWSPClient_AMQP 客户端遵循 OASIS 规范，实现了完整的 **AMQP 0.9.1** 协议。客户端支持纯 TCP 和 WebSocket 连接，也支持 TLS（安全）连接。

连接

AMQP 0.9.1 协议定义了频道的概念，允许您通过多个虚拟频道共享单个套接字连接，客户端实现了一个内部线程，该线程读取接收到的字节并将每条消息分发到正确的频道（每个频道已在其自己的线程中运行），因此，如果您运行带有 5 个频道的 AMQP 连接，客户端将运行 6 个线程（5 个处理每个频道数据的线程和 1 个处理连接数据的线程）。

在连接到 AMQP 服务器之前，请配置 AMQP 协议的以下属性

- **AMQPOptions.Locale**：要使用的消息区域设置，这是一个协商值，因此可能与服务器支持的区域设置不同。默认值为 "en_US"。
- **AMQPOptions.MaxChannels**：可打开的最大通道数，这是一个协商值，可能与服务器配置不同。默认值为 65535。
- **AMQPOptions.MaxFrameSize**：AMQP 帧的最大字节数，这是一个协商值，因此可能与服务器配置有所不同。默认值为 2147483647。
- **AMQPOptions.VirtualHost**：虚拟主机名称，默认值为 "/"。

连接到服务器之前也可以配置 AMQP 心跳，您可以启用或禁用心跳。

- **HeartBeat.Enabled**：如果客户端支持心跳，请设为 true。
- **HeartBeat.Interval**：所需间隔（以秒为单位）。

AMQP 客户端配置完成后，将其附加到 **TsgcWebSocketClient**，现在您可以配置服务器连接属性以连接到 AMQP 服务器。

如果使用的是纯 TCP 连接而非 WebSocket 连接，请将属性值 **Specifications.RFC6455** 设置为 False。

```
TsgcWSPClient_AMQP *oAMQP = new TsgcWSPClient_AMQP();
oAMQP->AMQPOptions->Locale = "en_US";
oAMQP->AMQPOptions->MaxChannels = 100;
oAMQP->AMQPOptions->MaxFrameSize = 16384;
oAMQP->AMQPOptions->VirtualHost = "/";
oAMQP->HeartBeat->Enabled = true;
oAMQP->HeartBeat->Interval = 60;

TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
oAMQP->Client = oClient;
oClient->Specifications->RFC6455 = false;
oClient->Host = "www.esegece.com";
oClient->Port = 5672;
oClient->Active = true;
```

通道

AMQP 客户端连接后，可以打开第一个通道。

```
oAMQP->OpenChannel("channel_name");
```

交易所

当通道打开时，客户端可以声明新的交换机、验证它们是否存在.....使用方法 **DeclareExchange** 声明新的交换机。

```
oAMQP->DeclareExchange("channel_name", "exchange_name");
```

队列

当频道打开后，客户端可以声明新队列，验证队列是否存在.....使用 **DeclareQueue** 方法声明新队列。队列默认不由服务器提供（与交换机不同），因此始终需要声明新队列（除非其他客户端已经创建了队列）。

```
oAMQP->DeclareQueue("channel_name", "queue_name");
```

绑定队列

配置好交换机和队列后，您可能需要将队列绑定到交换机，这样交换机便可知道哪些消息将被分发到哪些队列。

AMQP 服务器自动使用队列名称作为路由键将队列绑定到 "direct" 交换机。这允许您无需声明绑定即可向特定队列发送消息（只需调用 **PublishMessage** 方法，将 **Exchange** 参数设为空值，并在 **RoutingKey** 参数中填写队列名称）。

```
oAMQP->BindQueue("channel_name", "queue_name", "exchange_name", "routing_key");
```

发送消息

调用方法 **PublishMessage** 发布一条新的 AMQP 消息。该方法允许您发布 **String** 或 **TStream** 消息。

```
oAMQP->PublishMessage("channel_name", "exchange_name", "routing_key", "Hello from sgcWebSockets!!!");
```

接收消息

AMQP 允许您以 2 种模式接收消息：

- **客户端请求**：使用 **GetMessage** 方法。如果队列中没有消息，则会调用事件 **OnAMQPBasicGetEmpty**。
- **由服务器推送**：使用 **Consume** 方法。

客户端请求

```
oAMQP->GetMessage("channel_name", "queue_name");

void OnAMQPGetOk(TObject *Sender, const string aChannel,
const TsgcAMQPFramePayload_Method_BasicGetOk *aGetOk, const TsgcAMQPMessageContent *aContent)
{
    DoLog("#AMQP_basic_GetOk: " + aChannel + " " + IntToStr(aGetOk->MessageCount) + " " + aContent->Body->AsStr
}
```

由服务器推送

```
oAMQP->Consume("channel_name", "queue_name");

void OnAMQPGetOk(TObject *Sender, const string aChannel,
const TsgcAMQPFramePayload_Method_BasicGetOk *aGetOk, const TsgcAMQPMessageContent *aContent)
{
    DoLog("#AMQP_basic_GetOk: " + aChannel + " " + IntToStr(aGetOk->MessageCount) + " " + aContent->Body->AsStr
}
```

参考

- 属性 — 组件上可用的 6 个属性。
- 方法 — 41 个公共方法。
- 事件 — 25 个可处理的事件。

连接 | 客户端 AMQP 连接

要连接到 AMQP 服务器，必须先创建 `TsgcWebSocketClient` 和 `TsgcWSPClient_AMQP`，然后将 AMQP 组件附加到 WebSocket 客户端。

基本用法

无需身份验证即可连接到 AMQP 服务器。定义 `AMQPOptions` 属性值、虚拟主机，然后在 `TsgcWebSocketClient` 中设置服务器的 Host 和 Port。

如果您使用 TCP 普通连接，请将 `TsgcWebSocketClient` 属性 `Specifications.RFC6455` 设为 `false`。

```
oAMQP = new TsgcWSPClient_AMQP();
oAMQP->AMQPOptions->Locale = "en_US";
oAMQP->AMQPOptions->MaxChannels = 100;
oAMQP->AMQPOptions->MaxFrameSize = 16384;
oAMQP->AMQPOptions->VirtualHost = "/";
oAMQP->HeartBeat->Enabled = true;
oAMQP->HeartBeat->Interval = 60;

oClient = new TsgcWebSocketClient();
oAMQP->Client = oClient;
oClient->Specifications->RFC6455 = false;
oClient->Host = "www.esegece.com";
oClient->Port = 5672;
oClient->Active = true;
```

身份验证

如果服务器需要身份验证，请使用事件 `OnAMQPAuthentication` 选择身份验证机制（如需要）并设置用户名/密码。

```
oAMQP = new TsgcWSPClient_AMQP();
oAMQP->AMQPOptions->Locale = "en_US";
oAMQP->AMQPOptions->MaxChannels = 100;
oAMQP->AMQPOptions->MaxFrameSize = 16384;
oAMQP->AMQPOptions->VirtualHost = "/";
oAMQP->HeartBeat->Enabled = true;
oAMQP->HeartBeat->Interval = 60;

oClient = new TsgcWebSocketClient();
oAMQP->Client = oClient;
oClient->Specifications->RFC6455 = false;
oClient->Host = "www.esegece.com";
oClient->Port = 5672;
oClient->Active = true;

void OnAMQPAuthentication(TObject *Sender, TsgcAMQPAuthentications *aMechanisms, TsgcAMQPAuthentication &Mechanism,
String &User, String &Password)
{
    User = "user_value";
    Password = "password_value";
}
```


连接 | 客户端 AMQP 断开连接

客户端可以使用以下方法断开当前活动连接：

发送关闭原因

AMQP 客户端可通知服务器连接即将关闭，并提供关闭原因。使用 Close 方法向服务器请求关闭连接。

```
oAMQP.Close(541, "Internal Error");
```

关闭套接字连接

只需将 [TsgcWebSocketClient](#) 的 Active 属性设为 False。您可以阅读更多关于 [关闭连接](#) 的内容。

命令 | AMQP 通道

AMQP 是一种多信道协议。信道提供了一种将重量级 TCP/IP 连接复用为多个轻量级连接的方式。这使协议更具“防火墙友好性”，因为端口使用是可预测的。这也意味着流量整形和其他网络 QoS 功能可以轻松应用。

每个频道在自己的线程中运行，因此每次收到新消息时，客户端首先识别频道，然后将消息放入队列，由频道线程处理。

频道生命周期如下：

1. 客户端打开一个新通道（Open）。
2. 服务器确认新通道已就绪（Open-Ok）。
3. 客户端和服务端按需使用该通道。
4. 一方（客户端或服务端）关闭通道（Close）。
5. 另一个对端握手确认通道关闭（Close-Ok）。

打开频道

要创建新频道，只需调用 **OpenChannel** 方法，并将频道名称作为参数传入。事件 **OnAMQPChannelOpen** 作为服务器发送的确认触发，表明频道已打开。

```
AMQP->OpenChannel("channel_name");

private void OnAMQPChannelOpen(TObject *Sender, const string aChannel)
{
    DoLog("#AMQP_channel_open: " + aChannel);
}
```

也可以通过调用方法 **OpenChannelEx** 进行同步调用，该方法在通道成功打开时返回 true，若未收到服务器确认则返回 false。

```
if AMQP->OpenChannelEx("channel_name")
{
    DoLog("#AMQP_channel_open channel_name");
}
else
{
    DoLog("#AMQP_channel_open_error");
}
```

关闭频道

要关闭现有通道，请调用 **CloseChannel** 方法并传入通道名称作为参数。当客户端收到通道已关闭的确认时，将调用 **OnAMQPChannelClose** 事件。

可通过调用方法 **CloseChannelEx** 执行同步调用，该方法在通道已关闭时返回 true，若未收到服务器确认则返回 false。

通道流

流量控制是一种紧急措施，用于停止来自对端的消息流。它在客户端和服务端之间的工作方式相同，由 **EnableChannel / DisableChannel** 命令实现。流量控制是唯一能够阻止过度生产的发布者的机制。

要禁用频道的流量，请调用方法 **DisableChannel**，当客户端收到频道流量已禁用的确认时，将调用事件 **OnAMQPChannelFlow**。
启用通道流时同样如此，调用方法 **EnableChannel**，当客户端收到通道流已启用的确认时，将调用事件 **OnAMQPChannelFlow**。

同步请求可通过 **EnableChannelEx** 和 **DisableChannelEx** 函数获得。

命令 | AMQP 交换机

Exchange 类允许应用程序管理服务器上的 Exchange。该类使应用程序能够编写自己的路由规则（而不是依赖某些配置界面）。注意：大多数应用程序不需要如此高级的功能，旧版中间件也不太可能支持此语义。

交换机的生命周期如下：

1. 客户端请求服务器确认交换机存在（声明）。客户端可以细化为“如果不存在则创建交换机”，或“如果不存在请警告但不要创建”。
2. 客户端向交换机发布消息。
3. 客户端可选择删除交换机（Delete）。

声明交换机

此方法创建一个新的 Exchange 或验证 Exchange 是否已存在。该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **ExchangeName**：交换机的名称，长度不得超过 255 个字符，且不能以“amq.”开头（除非 passive 参数为 true）。
- **ExchangeType**：交换机类型，所有 AMQP 服务器均支持“direct”和“fanout”交换机。请查阅服务器文档以了解支持哪些交换机类型。
- **Passive**：如果 Passive 为 True，服务器只验证交换器是否已声明。如果 Passive 为 False 且交换器不存在，服务器将创建一个新的。
- **Durable**：如果为 true，服务器启动时将重新创建该 exchange。如果为 false，服务器停止时将删除该 exchange。
- **AutoDelete**：如果为 true，则当所有队列解绑后，交换机将被删除。
- **Internal**：始终为 false。
- **NoWait**：若为 true，服务器不向客户端发送确认。
- **Arguments**：包含自定义参数的字符串，值必须以 JSON 字符串格式传递，示例：{"x-dead-letter-exchange":"my-dlx"}。

要声明新交换机，请调用 **DeclareExchange** 方法，并传入频道名称、交换机名称和交换机类型作为参数。服务器将触发 **OnAMQPExchangeDeclare** 事件作为确认，表明交换机已声明。

```
AMQP->DeclareExchange("channel_name", "exchange_name", "direct");

private void OnAMQPExchangeDeclare(TObject *Sender, const string aChannel, const string aExchange)
{
    DoLog("#AMQP_exchange_declare: [" + aChannel + "] " + aExchange);
}
```

也可以通过调用方法 **DeclareExchangeEx** 进行同步调用，该方法在交换器已声明时返回 True，在未收到服务器确认时返回 False。

```
if AMQP->DeclareExchangeEx("channel_name", "exchange_name", "direct")
{
    DoLog("#AMQP_exchange_declare: [" + aChannel + "] " + aExchange);
}
else
{
    DoLog("#AMQP_exchange_declare_error");
}
```

删除交换机

此方法用于删除现有的交换器。该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。

- **ExchangeName**：交换机的名称，长度不得超过 255 个字符，且不能以"amq."开头（除非 passive 参数为 true）。
- **IfUnused**：仅当没有任何队列绑定到该 Exchange 时，服务器才删除它。
- **NoWait**：若为 true，服务器不向客户端发送确认。

要删除现有交易所，请调用方法 **DeleteExchange**，并将通道名称和交易所名称作为参数传递。服务器发送确认后会触发事件 **OnAMQPExchangeDelete**，表明交易所已被删除。

也可以通过调用方法 **DeleteExchangeEx** 进行同步调用，如果 Exchange 已被删除则返回 true，如果未收到服务器确认则返回 false。

命令 | AMQP 队列

队列类允许应用程序在服务器上管理消息队列。对于几乎所有消费消息的应用程序来说，这都是基本步骤，至少可以验证预期的消息队列实际上是存在的。

持久消息队列的生命周期相当简单：

1. 客户端断言消息队列存在（Declare，带 "passive" 参数）。
2. 服务器确认消息队列已存在（Declare-Ok）。
3. 客户端从消息队列中读取消息。

临时消息队列的生命周期更为有趣：

1. 客户端创建消息队列（Declare，通常不指定队列名称，由服务器分配名称）。服务器确认（Declare-Ok）。
2. 客户端在消息队列上启动一个消费者。消费者的具体功能由 Basic 类定义。
3. 客户端取消消费者，可以是显式取消，也可以通过关闭通道和/或连接来实现。
4. 当消息队列中最后一个消费者消失后，经过一段礼貌的超时，服务器将删除消息队列。

AMQP 将主题订阅的投递机制实现为消息队列。这支持一些有趣的结构，例如在一组协作订阅者之间对订阅进行负载均衡。

订阅的生命周期包含一个额外的绑定阶段：

1. 客户端创建消息队列（Declare），服务器确认（Declare-Ok）。
2. 客户端将消息队列绑定到主题交换机（Bind），服务器确认（Bind-Ok）。
3. 客户端像前面的示例一样使用消息队列。

声明队列

此方法创建新队列或验证队列是否已存在。该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **QueueName**：队列的名称，长度不超过 255 个字符，且不能以 "amq." 开头（当 passive 参数为 true 时除外）。
- **Passive**：如果 passive 为 true，服务器只验证队列是否已声明。如果 passive 为 false 且队列不存在，服务器将创建一个新队列。
- **Durable**：若为 true，服务器启动时将重新创建队列；若为 false，服务器停止时将删除队列。
- **Exclusive**：若为 True，表示队列仅由当前连接访问。
- **AutoDelete**：若为 true，当所有消费者停止使用该队列时，队列将被删除。
- **NoWait**：若为 true，服务器不向客户端发送确认。
- **Arguments**：包含自定义参数的字符串，值必须以 JSON 字符串格式传递，示例：{"x-dead-letter-exchange":"my-dlx"}。

要声明新队列，只需调用方法 **DeclareQueue** 并传入通道名称和队列名称作为参数。事件 **OnAMQPQueueDeclare** 将作为服务器确认交换机已声明的回调被触发。

```
AMQP->DeclareQueue("channel_name", "queue_name");

private void OnAMQPExchangeDeclare(TObject *Sender, const string aChannel, const string aQueue,
    int aMessageCount, int aConsumerCount)
{
    DoLog("#AMQP_queue_declare: [" + aChannel + "] " + aQueue));
}
```

也可以通过调用方法 **DeclareQueueEx** 进行同步调用，如果队列已被声明则返回 true，如果没有收到服务器确认则返回 false。

```
if AMQP->DeclareQueueEx("channel_name", "queue_name")
{
    DoLog("#AMQP_queue_declare: [" + aChannel + "]" + aQueue);
}
else
{
    DoLog("#AMQP_queue_declare_error");
}
```

删除队列

此方法用于删除现有队列。该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **QueueName**：队列的名称，长度不超过 255 个字符，且不能以"amq."开头（当 **passive** 参数为 **true** 时除外）。
- **IfUnused**：只有在没有任何消费者附加到队列时，服务器才会删除该队列。
- **IfEmpty**：仅当队列中没有消息时，服务器才删除该队列。
- **NoWait**：若为 **true**，服务器不向客户端发送确认。

要删除现有队列，请调用方法 **DeleteQueue**，并将通道名称和队列名称作为参数传递。服务器发送确认后会触发事件 **OnAMQPQueueDelete**，表明队列已被删除。

也可以调用 **DeleteQueueEx** 方法进行同步调用，该方法在队列已删除时返回 **true**，在未收到服务器确认时返回 **false**。

绑定队列

此方法用于将队列绑定到交换机。交换机使用绑定来确定将使用哪些队列来路由消息。

所有 AMQP 服务器都会自动将所有队列绑定到默认交换机（这是一个没有名称的"direct"交换机），使用队列名称作为绑定路由键。这允许您无需声明绑定就可以向特定队列发送消息。只需调用方法 **PublishMessage**，将交换机名称传为空值，并将 **RoutingKey** 设置为 **队列名称** 的值即可。

该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **QueueName**：队列的名称，长度不超过 255 个字符，且不能以"amq."开头（当 **passive** 参数为 **true** 时除外）。
- **ExchangeName**：交换机的名称，长度不得超过 255 个字符，且不能以"amq."开头（除非 **passive** 参数为 **true**）。
- **RoutingKey**：绑定的路由键。
- **NoWait**：若为 **true**，服务器不向客户端发送确认。
- **Arguments**：包含自定义参数的字符串，值必须以 JSON 字符串格式传递，示例：**{"x-dead-letter-exchange":"my-dlx"}**。

要将队列绑定到交换机，请调用方法 **BindQueue** 并传入通道名称、队列名称、交换机和路由键作为参数。服务器发送队列已绑定的确认时，将触发事件 **OnAMQPQueueBind**。

```
AMQP->BindQueueEx("channel_name", "queue_name", "exchange_name", "routing_key");

private void OnAMQPQueueBind(TObject *Sender, const string aChannel, const string aQueue,
    const string aExchange)
{
    DoLog("#AMQP_queue_bind: [" + aChannel + "]" + aQueue + " -->-- " + aExchange);
}
```

也可以调用 **BindQueueEx** 方法进行同步调用，该方法在队列已绑定时返回 **true**，在未收到服务器确认时返回 **false**。

```
if AMQP->BindQueueEx("channel_name", "queue_name", "exchange_name", "routing_key")
{
```

```

    DoLog("#AMQP_queue_bind: [" + aChannel + "] " + aQueue + " -->-- " + aExchange);
}
else
{
    DoLog("#AMQP_queue_bind_error");
}

```

解绑队列

此方法删除现有的队列绑定。

该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **QueueName**：队列的名称，长度不超过 255 个字符，且不能以"amq."开头（当 passive 参数为 true 时除外）。
- **ExchangeName**：交换机的名称，长度不得超过 255 个字符，且不能以"amq."开头（除非 passive 参数为 true）。
- **RoutingKey**：绑定的路由键。

要解除队列绑定，只需调用 **UnBindQueue** 方法，并传入频道名称、队列名称、交换机和路由键作为参数。服务器将触发 **OnAMQPQueueUnBind** 事件作为确认，表明队列已解除绑定。

也可以通过调用方法 **UnBindQueueEx** 进行同步调用，如果队列已解绑则返回 True，如果未收到服务器确认则返回 False。

清除队列

此方法清除队列的所有消息。已发送但等待确认的消息不受影响。

该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **QueueName**：队列的名称，长度不超过 255 个字符，且不能以"amq."开头（当 passive 参数为 true 时除外）。
- **NoWait**：若为 true，服务器不向客户端发送确认。

要清空队列，只需调用 **PurgeQueue** 方法并传入信道名称和队列名称作为参数。服务器确认队列已被清空后，将触发 **OnAMQPQueuePurge** 事件。

也可通过调用方法 **PurgeQueueEx** 进行同步调用，该方法在队列已被清空时返回 True，在未收到服务器确认时返回 False。

命令 | AMQP 发布消息

发布消息

方法 `PublishMessages` 用于向 AMQP 服务器发送消息。

AMQP 服务器自动使用队列名称作为路由键将队列绑定到 "direct" 交换机。这允许您无需声明绑定即可向特定队列发送消息（只需调用 `PublishMessage` 方法，将 `Exchange` 参数设为空值，并在 `RoutingKey` 参数中填写队列名称）。

该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **ExchangeName**：交换机名称，长度不能超过 255 个字符，且不能以 "amq." 开头（除非 `passive` 参数为 `true`）。
- **RoutingKey**：绑定的路由键名称。
- **Mandatory**：若为 `true`，且消息无法路由到任何队列，则服务器将退回该消息，并触发事件 `OnAMQPBasicReturn`。
- **Immediate**：如果为 `true`，且消息无法路由到任何队列，服务器将返回该消息并触发 `OnAMQPBasicReturn` 事件。

```
AMQP->PublishMessage("channel_name", "exchange_name", "routing_key", "Hello from sgcWebSockets!!!");

private void OnAMQPBasicReturn(TObject *Sender, const string aChannel,
    const TsgcAMQPFramePayload_Method_BasicReturn *aReturn,
    const TsgcAMQPMessageContent *aContent)
{
    DoLog("#AMQP_basic_return: " + aChannel + " " + IntToStr(aReturn->ReplyCode) + " " + aReturn->ReplyText + " " +
}
```

发布确认

网络在发布消息时可能会出现故障，保证消息不丢失的唯一方法是使用事务，然后对每条消息选择事务、发送消息并提交。成功事务的确认在触发事件 `OnAMQPTransactionOk` 时收到。

AMQP 消费消息

消费者从队列中消费消息。为了消费消息，必须存在一个队列。添加新消费者后，假设队列中已有就绪消息，投递将立即开始。

消费者注册时目标队列可以为空。在这种情况下，当新消息入队后，首次投递才会发生。

消费消息是一项**异步**任务，这意味着每当有新消息可以投递到消费者队列时，服务器会自动将其推送给客户端。您可以通过阅读另一种[同步接收消息](#)的方法。

消费

方法 **Consume** 在队列中创建一个新消费者，每当有新消息时，该消息将自动传送给消费者客户端。

该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **QueueName**：队列名称，长度不得超过 255 个字符，且不得以 "amq." 开头（passive 参数为 true 时除外）。
- **ConsumerTag**：消费者的名称，必须唯一。如果未设置，则服务器会创建一个新名称。
- **NoLocal**：如果为 true，则消费者不会消费在同一频道上发布的消息。
- **NoAck**：若为 true，表示服务器不需要对每条已投递的消息进行确认。
- **Exclusive**：若为 True，则阻止其他消费者消费此队列中的消息。
- **NoWait**：如果为 True，服务器不会向客户端发送确认。
- **Arguments**：包含自定义参数的字符串，值必须以 JSON 字符串格式传递，示例：{"x-dead-letter-exchange":"my-dlx"}。

消息通过 **OnAMQPBasicDeliver** 事件传递。

```
AMQP->Consume("channel_name", "queue_name", "consumer_tag");

void OnAMQPBasicDeliver(TObject *Sender, const string aChannel,
    const TsgcAMQPFramePayload_Method_BasicDeliver *aDeliver,
    const TsgcAMQPMessageContent *aContent)
{
    DoLog("#AMQP_basic_deliver: " + aChannel + " " + aDeliver->ConsumerTag + " " +
        " " + aContent->Body->AsString);
}
```

也可以直接调用 **ConsumeEx** 方法进行同步调用，该方法在消费者已创建时返回 true，在未收到服务器确认时返回 false。

取消消费

此方法用于取消现有的消费者队列。

该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **ConsumerTag**：消费者的名称。
- **NoWait**：如果为 True，服务器不会向客户端发送确认。

```
AMQP->CancelConsume("channel_name", "consumer_tag");

private void OnAMQPBasicCancelConsume(TObject *Sender, const string aChannel, const string aConsumerTag)
{
    DoLog("#AMQP_basic_cancel_consume: " + aChannel + " " + aConsumerTag);
}
```

也可通过调用方法 **CancelConsumeEx** 进行同步调用，该方法在消费者已取消时返回 True，在未收到服务器确认时返回 False。

命令 | AMQP 获取消息

获取消息是一个 **同步** 任务，这意味着是客户端向服务器询问队列中是否有消息。您可以阅读[异步接收消息](#)的替代方法。

获取消息

GetMessage 方法向 AMQP 服务器发送请求，询问队列中是否有可用消息。如果有消息，将通过 **OnAMQPBasicGetOk** 事件分发；如果队列为空，则调用事件 **OnAMQPBasicGetEmpty**。

该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **QueueName**：队列的名称，长度不超过 255 个字符，且不能以"amq."开头（当 **passive** 参数为 **true** 时除外）。
- **NoWait**：如果为 **True**，服务器不会向客户端发送确认。

```
AMQP->GetMessage("channel_name", "queue_name");

private void OnAMQPBasicGetOk(TObject *Sender, const string aChannel,
    const TsgcAMQPFramePayload_Method_BasicGetOk *aGetOk,
    const TsgcAMQPMessageContent *aContent)
{
    DoLog("#AMQP_basic_GetOk: " + aChannel + " " + IntToStr(aGetOk->MessageCount) + " " + aContent->Body->AsString)
}

private void OnAMQPBasicGetEmpty(TObject *Sender, const string aChannel)
{
    DoLog("#AMQP_basic_GetEmpty: " + aChannel);
}
```

可通过调用方法 **GetMessageEx** 执行同步调用，若队列中有可用消息则返回 **true**，否则返回 **false**。

命令 | AMQP QoS

AMQP 允许您设置 QoS 级别，以限制服务器在等待消息确认之前向客户端发送的消息数量。

设置 QoS

方法 **SetQoS** 用于限制服务器发送给 AMQP 客户端的消息数量。

该方法具有以下参数：

- **ChannelName**：频道名称（调用此方法前必须已打开该频道）。
- **PrefetchSize**：这是以字节为单位的窗口大小。如果当前所有未确认消息的总大小加上下一条待发送消息的大小超过 **PrefetchSize** 参数，服务器将不会向客户端发送消息。值为零表示没有限制。
- **PrefetchCount**：是已发送但尚未确认的未确认消息的最大数量，如果数量超过此值，服务器将停止向客户端发送消息。
- **Global**：若为 true，QoS 适用于连接的所有现有和新消费者。若为 false，QoS 适用于通道的所有现有和新消费者。

服务器的响应通过 **OnAMQPBasicQoS** 事件接收。

```
AMQP->SetQoS("channel_name", 1024000, 100, false);

private void OnAMQPBasicQoS(TObject *Sender, const string aChannel,
    const TsgcAMQPFramePayload_Method_BasicQoS *aQoS)
{
    DoLog("#AMQP_basic_qos: " + aChannel + " " + IntToStr(aQoS->PrefetchSize) + " "
        + IntToStr(aQoS->PrefetchCount) + " " + BoolToStr(aQoS->Global));
}
```

同步调用只需调用 **SetQoSEx** 方法即可，如果请求已被处理则返回 True，否则返回 False。

命令 | AMQP 事务

AMQP 支持两种类型的事务：

1. 自动事务，其中每条已发布的消息和确认均作为独立事务处理。
2. 服务器本地事务，服务器将缓冲已发布的消息和确认，并根据客户端的请求按需提交。

Transaction 类 ("tx") 允许应用程序访问第二种类型，即服务器事务。此类的语义如下：

1. 应用程序在每个需要事务的通道中请求服务器事务 (Select)。
2. 应用程序执行工作 (Publish、Ack)。
3. 应用程序提交或回滚工作 (Commit、Roll-back)。
4. 应用程序持续运行。

事务涵盖已发布的内容和确认，而不涵盖已传递的内容。因此，回滚不会重新排队或重新传递任何消息，客户端有权在后续事务中确认这些消息。

事务方法允许将发布和确认操作批量处理为原子工作单元。其意图是事务中发出的所有发布和确认请求要么全部成功完成，要么全部不成功。

开始事务

StartTransaction 方法在服务器中启动一个新事务，客户端在使用 Commit 或 Rollback 方法之前，必须在频道上至少调用一次此方法。当服务器确认使用事务时，将触发事件 **OnAMQPTransactionOk**。

```
AMQP->StartTransaction("channel_name");
```

同步调用可通过调用方法 **StartTransactionEx** 完成，若请求已成功处理则该方法返回 true，否则返回 false。

提交事务

此方法提交当前事务中执行的所有消息发布和确认。提交后立即开始新的事务。当服务器确认事务使用时，将触发事件 **OnAMQPTransactionOk**。

```
AMQP->CommitTransaction("channel_name");
```

也可以通过调用方法 **CommitTransactionEx** 进行同步调用，如果请求已被处理则返回 true，否则结果为 false。

回滚事务

此方法放弃当前事务中执行的所有消息发布和确认操作。回滚后立即开始新事务。请注意，未确认的消息不会因回滚而自动重新投递；如有需要，应显式发出恢复调用。服务器确认事务使用后，将引发事件 **OnAMQPTransactionOk**。

```
AMQP->RollbackTransaction("channel_name");
```

也可以通过调用方法 **RollbackTransactionEx** 进行同步调用，该方法在请求处理成功时返回 true，否则返回 false。

协议 Kafka

Apache Kafka 是一个开源的分布式事件流平台，用于发布、存储和消费记录流。生产者将消息写入主题，代理将其存储在分区的仅追加日志中，消费者按顺序读取消息，既可以单独消费，也可以作为协调消费组的一部分。

Kafka 在 TCP 上使用二进制应用层协议。每个主题被分割为一个或多个分区，分区中的每条记录由单调递增的偏移量标识，代理保留这些记录，以便消费者可以按照自己的节奏读取并在需要时重放历史记录。这使 Kafka 非常适合高吞吐量消息传递、事件溯源、日志聚合和流处理。

功能特性

sgcWebSockets Kafka 客户端直接通过原生 Kafka 协议与代理通信，无需 REST 代理或外部库。

- 向任何主题和分区发布（生产）记录，带有键、值和可配置的确认。
- 使用托管消费者组或不使用组（直接读取所有分区）消费记录。
- 消费者组协调：协调器发现、加入/同步、分区分配以及偏移量提交/获取。
- 偏移量管理：查询最早、最新和已提交的偏移量。
- 主题管理：创建和删除主题、请求集群元数据以及列出消费者组。
- 读写当前 Kafka 代理使用的 v2 记录批次格式。

支持的版本

客户端使用 **Apache Kafka 0.11** 中引入的 v2 记录批次格式，并已针对 **Apache Kafka 3.x** 进行了测试。默认 broker 端口为 9092。

组件

TsgcWSPClient_Kafka：实现 **Apache Kafka** 协议的客户端组件。

最常见的用途

- 连接
 - [Client Kafka Connect](#)
 - [Client Kafka Disconnect](#)
- 命令
 - [Kafka 生产消息](#)
 - [Kafka 消费消息](#)
 - [Kafka 消费者组](#)
 - [Kafka 主题](#)
 - [Kafka 偏移量](#)

TsgcWSPClient_Kafka

Apache Kafka 客户端。

简介

TsgcWSPClient_Kafka 客户端实现了 **Apache Kafka** 协议，通过普通 TCP 连接上的原生 Kafka 二进制协议直接与代理通信。它无需 REST 代理或外部库，即可生产和消费记录、管理消费者组、管理主题和查询偏移量。

连接

Kafka 客户端是一个运行在 **TsgcWebSocketClient** 之上的协议组件。由于 Kafka 不是 WebSocket 协议，请将 **Specifications.RFC6455** 设置为 **false**，使客户端以普通 TCP 套接字方式连接。然后将客户端分配给 Kafka 组件并设置代理的 **Host** 和 **Port**（默认 9092）。

生产者和消费者的行为通过 **KafkaOptions** 属性进行配置：

- **KafkaOptions.ClientId**：向代理标识您的应用程序。
- **KafkaOptions.Producer.Acks**：所需的确认数，可选 **kafkaAcksNone**、**kafkaAcksLeader** 或 **kafkaAcksAll**。
- **KafkaOptions.Producer.TimeoutMs**：代理等待所需确认的时长。
- **KafkaOptions.Consumer.GroupId**：消费者组名称。留空则表示不使用组进行消费。
- **KafkaOptions.Consumer.OffsetReset**：**kafkaOffsetEarliest** 或 **kafkaOffsetLatest**，表示全新消费者组的起始读取位置。

```
TsgcWSPClient_Kafka *oKafka = new TsgcWSPClient_Kafka();
oKafka->KafkaOptions->ClientId = "my-cbuilder-app";
oKafka->OnKafkaConnect = OnKafkaConnect;
oKafka->OnKafkaMessage = OnKafkaMessage;
oKafka->OnKafkaProduce = OnKafkaProduce;

TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
oKafka->Client = oClient;
oClient->Specifications->RFC6455 = false;
oClient->Host = "127.0.0.1";
oClient->Port = 9092;
oClient->Active = true;
```

生产消息

调用方法 **Produce** 以发布记录。每次发布的结果通过 **OnKafkaProduce** 事件报告，包括记录存储时的偏移量。

```
oKafka->KafkaOptions->Producer->Acks = kafkaAcksLeader;
oKafka->Produce("my-topic", "Hello Kafka", "key-1");
```

消费消息

设置 **GroupId** 和 **OffsetReset**，订阅一个或多个主题，然后调用 **Poll** 获取可用记录。Poll 返回 **TsgcKafkaMessages** 列表，并对每条记录触发 **OnKafkaMessage** 事件。重复调用 Poll（例如通过定时器）以持续接收消息。

```
oKafka->KafkaOptions->Consumer->GroupId = "my-group";
oKafka->KafkaOptions->Consumer->OffsetReset = kafkaOffsetEarliest;
String topics[1] = { "my-topic" };
oKafka->Subscribe(topics, 0);

TsgcKafkaMessages *oMessages = oKafka->Poll(1000);
try {
    for (int i = 0; i < oMessages->Count; i++)
        DoLog(oMessages->Items[i]->GetValueString());
} __finally {
```



```
oMessages->Free();  
}
```

参考

- 属性 — 组件上提供 3 个属性。
- 方法 — 18 个公共方法。
- 事件——可处理 6 个事件。

连接 | 客户端 Kafka Connect

要连接到 Apache Kafka 代理，必须先创建一个 [TsgcWebSocketClient](#) 和一个 [TsgcWSPClient_Kafka](#)，然后将 Kafka 组件附加到 WebSocket 客户端。

Kafka 使用纯 TCP 连接，因此您必须将 `TsgcWebSocketClient` 属性 `Specifications.RFC6455` 设置为 `false`。设置代理的 Host 和 Port（Kafka 默认端口为 9092），可选设置 `KafkaOptions.ClientId`，然后将 `Active` 设置为 `true` 以启动连接。

当与代理建立连接时，触发 **OnKafkaConnect** 事件。

基本用法

```
oKafka = new TsgcWSPClient_Kafka();
oKafka->KafkaOptions->ClientId = "sgc-client";

oClient = new TsgcWebSocketClient();
oKafka->Client = oClient;
oClient->Specifications->RFC6455 = false;
oClient->Host = "www.esegece.com";
oClient->Port = 9092;
oClient->Active = true;

void OnKafkaConnect(TObject *Sender)
{
    ShowMessage("Connected to Kafka broker");
}
```

连接 | 客户端 Kafka 断开连接

要从 Apache Kafka 代理断开连接，请将 WebSocket 客户端属性 Active 设置为 false。这将关闭到代理的底层 TCP 连接。

当连接关闭时，将触发 **OnKafkaDisconnect** 事件。

基本用法

```
oClient->Active = false;  
  
void OnKafkaDisconnect(TObject *Sender)  
{  
    ShowMessage("Disconnected from Kafka broker");  
}
```

命令 | Kafka 生产消息

使用 **Produce(topic, value, key, partition)** 方法向 Kafka 主题发布记录。key 和 partition 参数为可选；未设置分区时，代理自行选择（例如通过对 key 取哈希）。调用 **ProduceBytes** 可发布二进制负载。

通过 **KafkaOptions.Producer.Acks** 配置投递保证：**kafkaAcksNone**（即发即忘）、**kafkaAcksLeader**（等待分区 leader）或 **kafkaAcksAll**（等待所有同步副本）。还可以设置 **Producer.TimeoutMs** 和 **Producer.Compression**（**kafkaCompressionNone** 或 **kafkaCompressionGzip**）。

记录发送后，**OnKafkaProduce** 事件将报告所生产记录的主题、分区、偏移量和错误代码。

基本用法

```
oKafka->KafkaOptions->Producer->Acks = kafkaAcksAll;
oKafka->KafkaOptions->Producer->TimeoutMs = 5000;
oKafka->KafkaOptions->Producer->Compression = kafkaCompressionGzip;

oKafka->Produce("my-topic", "hello world", "key1");

void OnKafkaProduce(TObject *Sender, const String Topic, int Partition, __int64 Offset, int ErrorCode)
{
    if (ErrorCode == 0)
        ShowMessage("Produced to " + Topic + " at offset " + IntToStr(Offset));
    else
        ShowMessage("Produce error " + IntToStr(ErrorCode));
}
```

Commands | Kafka 消费消息

要消费记录，请设置 `KafkaOptions.Consumer.GroupId` 和 `KafkaOptions.Consumer.OffsetReset` (`kafkaOffsetEarliest` 或 `kafkaOffsetLatest`)，然后使用一个或多个主题调用 **Subscribe([topic])**。订阅后，调用 **Poll(timeoutMs)** 以获取记录。Poll 返回一个 `TsgcKafkaMessages` 列表，并为每条记录触发一次 **OnKafkaMessage** 事件。通常需要反复调用 Poll（例如通过定时器），以持续消费新到达的记录。列表中的每条记录均为 `TsgcKafkaMessage`。使用 Count 遍历列表，并通过 `GetKeyString`、`GetValueString`、`Topic`、`Partition` 和 `Offset` 读取每条记录。

订阅和轮询

```
oKafka->KafkaOptions->Consumer->GroupId = "my-group";
oKafka->KafkaOptions->Consumer->OffsetReset = kafkaOffsetEarliest;
oKafka->Subscribe(ARRAYOFCONST("my-topic"));

// call repeatedly, e.g. from a TTimer::OnTimer
TsgcKafkaMessages *oMessages = oKafka->Poll(1000);
for (int i = 0; i < oMessages->Count; i++)
    ShowMessage(oMessages->Items[i]->GetKeyString() + " = " + oMessages->Items[i]->GetValueString());
```

OnKafkaMessage 事件

您可以在 `OnKafkaMessage` 事件中处理每条记录，而不必遍历 Poll 返回的列表。

```
void OnKafkaMessage(TObject *Sender, TsgcKafkaMessage *Message)
{
    ShowMessage(Message->Topic + ": " + Message->GetKeyString() + " = " + Message->GetValueString());
}
```

命令 | Kafka 消费者组

消费者组允许多个消费者共同完成读取主题的工作。代理按 GroupId 跟踪已提交的偏移量，并将主题分区分配给组的成员。

对于没有已提交偏移量的新消费者组，KafkaOptions.Consumer.OffsetReset 决定从哪里开始：kafkaOffsetEarliest 从分区起始位置读取，kafkaOffsetLatest 只读取新记录。一旦提交了偏移量，消费将从最后提交的位置恢复。

调用 **CommitSync** 提交最后一次 Poll 返回的记录的偏移量，以便消费组在下一个会话中从它们之后继续。如果您保持 GroupId 为空，客户端将直接消费所有分区，而不会加入消费组，也不会向代理提交偏移量。

基本用法

```
oKafka->KafkaOptions->Consumer->GroupId = "orders-group";
oKafka->KafkaOptions->Consumer->OffsetReset = kafkaOffsetEarliest;
oKafka->Subscribe(ARRAYOFCONST(("orders")));

TsgcKafkaMessages *oMessages = oKafka->Poll(1000);
if (oMessages->Count > 0)
    oKafka->CommitSync();
```

Commands | Kafka Topics

Kafka 客户端可以管理代理上的主题。使用 **CreateTopic(name, partitions, replication)** 以给定的分区数和复制因子创建新主题，使用 **DeleteTopic(name)** 删除主题。使用 **GetMetadata([topics])** 检索集群和主题信息，例如代理、主题及其分区布局。传入空数组以请求整个集群的元数据。

基本用法

```
// create a topic with 3 partitions and replication factor 1
oKafka->CreateTopic("my-topic", 3, 1);

// request metadata for the topic
oKafka->GetMetadata(ARRAYOFCONST(("my-topic")));

// delete the topic
oKafka->DeleteTopic("my-topic");
```

命令 | Kafka 偏移量

Kafka 客户端可以查询和设置给定主题和分区的偏移量。**GetEarliestOffset(topic, partition)** 返回最旧的可用偏移量，**GetLatestOffset(topic, partition)** 返回下一个待写入的偏移量，**GetCommittedOffset(topic, partition)** 返回消费者组最后提交的偏移量。三者均返回 `Int64` 类型。

使用 **CommitOffset(topic, partition, offset)** 为消费者组提交特定偏移量，以便在下一个会话中从该位置继续消费。

基本用法

```
__int64 vEarliest = oKafka->GetEarliestOffset("my-topic", 0);
__int64 vLatest = oKafka->GetLatestOffset("my-topic", 0);
__int64 vCommitted = oKafka->GetCommittedOffset("my-topic", 0);

// commit a specific offset for the consumer group
oKafka->CommitOffset("my-topic", 0, vLatest);
```


Protocol AMQP 1.0.0

AMQP（高级消息队列协议）1.0.0 是一种消息传递协议，专为分布式系统之间的可靠异步通信而设计。它以解耦的方式促进应用程序或组件之间的消息交换，使其无需直接依赖即可通信。以下是 AMQP 1.0.0 部分关键方面的技术说明：

- **面向消息的通信**：AMQP 1.0.0 以消息概念为核心。消息可以携带数据、指令或命令，是通信的基本单元。
- **消息代理**：该协议运行在代理消息模型上。代理（可以是服务器或中间实体）管理生产者和消费者之间消息的路由和投递。
- **队列与交换机**：队列是代理内部的存储实体，消息在其中临时存储。交换机定义了根据消息内容或路由键等条件，将消息从生产者路由到队列的规则。
- **地址和链路**：地址标识消息基础设施内的消息目的地。链路是发送方（生产者）和接收方（消费者）之间与特定地址关联的通信通道。
- **会话与连接**：会话为通信建立逻辑通道，允许在单个连接中传输多个消息流。连接管理客户端应用程序与消息代理之间的整体通信链路。
- **安全性**：AMQP 1.0.0 支持多种安全机制，包括认证和授权，以确保客户端与代理之间通信的安全。
- **传输无关性**：该协议设计为传输无关，即可在 TCP、TLS 或 WebSockets 等不同网络传输上运行，在部署时具有灵活性。
- **Flow Control**：AMQP 1.0.0 包含流量控制机制，允许消费者指示其在给定速率下处理传入消息的能力。这有助于防止大量消息压垮消费者。
- **错误处理**：该协议指定了处理错误的机制，包括消息的确认和拒绝，确保消息传递的健壮性和可靠性。
- **SASL 身份验证**：简单身份验证和安全层（SASL）用于验证客户端与代理之间连接的身份并确保其安全。

总体而言，AMQP 1.0.0 为不同软件组件和系统之间以松耦合方式进行通信提供了标准化且可互操作的方式，适用于各种分布式和企业级应用程序。

组件

TsgcWSPClient_AMQP1：实现 **AMQP 1.0.0** 协议的客户端组件。

最常见的用途

- **连接**
 - [客户端 AMQP1 连接](#)
 - [客户端 AMQP1 断开连接](#)
 - [客户端 AMQP1 空闲超时连接](#)
 - [客户端 AMQP1 连接状态](#)
 - [客户端 AMQP1 认证](#)
 - [客户端 AMQP1 Azure Service Bus](#)
- **命令**
 - [AMQP1 会话](#)
 - [AMQP1 链接](#)
 - [AMQP1 发送方链接](#)
 - [AMQP1 接收方链接](#)
 - [AMQP1 发送消息](#)
 - [AMQP1 读取消息](#)

TsgcWSPClient_AMQP1

AMQP 1.

简介

TsgcWSPClient_AMQP1 客户端遵循 OASIS 规范实现了 **AMQP 1.0.0** 协议。该客户端支持普通 TCP 和 WebSocket 连接，同时支持 TLS（安全）连接。

配置

AMQP 1.0.0 客户端的 `AMQPOptions` 属性用于配置连接。

- **ChannelMax**：channel-max 值是连接上可使用的最高通道编号。此值加一是连接上可同时活跃的最大会话数
- **ContainerId**：（可选）源容器的名称，在服务器中唯一标识连接。
- **CreditSize**：信用流的默认大小。
- **IdleTimeout**：当超过阈值时间内未收到任何帧时，本地对端触发超时。空闲超时以毫秒为单位，从收到最后一帧起计算。
- **MaxFrameSize**：可接受的最大帧大小。
- **MaxLinksPerSession**：每个会话的最大链接数。
- **WindowSize**：默认窗口大小。

AMQP 身份验证必须在 `Authentication` 属性中配置。

- **AuthType**：身份验证类型
 - **amqp1authNone**：未配置。
 - **amqp1authSASLAnonymous**：匿名身份验证
 - **amqp1authSASLPlain**：用户名/密码身份验证。此身份验证类型需要填写以下属性：
 - Username
 - Password
 - **amqp1authSASLExternal**：外部身份验证

连接

连接以客户端（通常是消息应用程序或服务）向服务器（消息代理）发起 TCP 连接开始。客户端连接到服务器端口，通常非 TLS 连接使用 5672，TLS 安全连接使用 5671。TCP 连接建立后，客户端和服务器协商将使用的 AMQP 协议版本。AMQP 1.0.0 支持多个版本，双方在协商过程中同意使用 1.0.0 版本。

协议协商完成后，客户端可能需要向服务器进行身份验证，具体取决于服务器的配置。身份验证机制可包括 SASL（简单身份验证和安全层）机制，如 PLAIN、EXTERNAL 或服务器支持的其他机制。

示例：连接到侦听安全端口 5671 并使用 SASL 凭证的 AMQP 服务器

```
// Creating AMQP client
oAMQP = new TsgcWSPClient_AMQP1(this);
// Setting AMQP authentication options
oAMQP->AMQPOptions->Authentication->AuthType = amqp1authSASLPlain;
oAMQP->AMQPOptions->Authentication->Username = L"sgc";
oAMQP->AMQPOptions->Authentication->Password = L"sgc";
// Creating WebSocket client
oClient = new TsgcWebSocketClient(this);
// Setting WebSocket specifications
oClient->Specifications->RFC6455 = false;
// Setting WebSocket client properties
```

```
oClient->Host = L"www.esegece.com";
oClient->Port = 5671;
oClient->TLS = true;
// Assigning WebSocket client to AMQP client
oAMQP->Client = oClient;
// Activating WebSocket client
oClient->Active = true;
```

会话

身份验证完成后，客户端打开一个 AMQP 会话。会话是客户端与服务器之间通信的逻辑上下文，用于将相关消息操作组合在一起。使用方法 **CreateSession** 创建新会话，该方法允许您设置会话名称，或留空让组件自动分配一个。

如果会话成功创建，事件 **OnAMQPSessionOpen** 将携带会话详情触发。

```
oAMQP->CreateSession("MySession");
oAMQP->OnAMQPSessionOpen = AMQP1AMQPSessionOpen;
void __fastcall TMyClass::AMQP1AMQPSessionOpen(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Fr
{
    ShowMessage("#session-open: " + aSession->Id);
}
```

链接

在会话中，客户端创建链路以与特定实体（如队列、主题或服务器提供的其他资源）通信。链路是用于发送和接收消息的双向通信通道。

该组件可作为发送节点和接收节点运行。允许为每个会话创建任意数量的链接，上限由 **MaxLinksPerSession** 属性设置。

发送方链路

要创建新的发送方链路，请使用 **CreateSenderLink** 方法并传入会话名称（可选传入发送方链路名称）。若链路创建成功，将触发 **OnAMQPLinkOpen** 事件。

```
oAMQP->CreateSenderLink("MySession", "MySenderLink");
oAMQP->OnAMQPLinkOpen = AMQP1AMQPLinkOpen;
void __fastcall TMyForm::AMQP1AMQPLinkOpen(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *
{
    ShowMessage("#link-open: " + aLink->Name);
}
```

接收链路

要创建新的接收方链路，请使用方法 **CreateReceiverLink**，并传入会话名称，以及可选的接收方链路名称。如果链路创建成功，将触发事件 **OnAMQPLinkOpen**。

```
oAMQP->CreateSenderLink("MySession", "MySenderLink");
oAMQP->OnAMQPLinkOpen = AMQP1AMQPLinkOpen;
void __fastcall TMyForm::AMQP1AMQPLinkOpen(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *
{
    ShowMessage("#link-open: " + aLink->Name);
}
```

```
}
```

发送消息

会话建立并创建链路后，客户端可以开始执行消息操作，例如向目的地发送消息。使用 `SendMessage` 方法通过发送方链路发送消息。

```
oAMQP->SendMessage("MySession", "MySenderLink", "My first AMQP Message");
```

接收消息

默认情况下，接收链路以**自动模式**创建，即每当有新消息到达时，都会自动投递给客户端。

如果接收方链接以**手动模式**创建，请使用同步方法 **GetMessage** 获取并等待新消息到达。

在自动和手动模式下，每次新消息到达时，都会触发 **OnAMQPMessage** 事件。

```
void __fastcall OnAMQPMessageEvent(TObject *Sender, TsgcAMQP1Session *const aSession,
    TsgcAMQP1ReceiverLink *const aLink, TsgcAMQP1Message *const aMessage,
    TsgcAMQP1MessageDeliveryState &DeliveryState)
{
    ShowMessage(aMessage->ApplicationData->AMQPValue->Value);
}
```

参考

- Properties — 组件上有 5 个可用属性。
- 方法 — 12 个公共方法。
- 事件 — 14 个可处理的事件。

连接 | 客户端 AMQP1 连接

要连接到 AMQP 服务器，必须先创建一个 [TsgcWebSocketClient](#) 和一个 [TsgcWSPClient_AMQP1](#)，然后将 AMQP1 组件附加到 WebSocket 客户端。

成功连接后，将触发 OnAMQPConnect 事件。

基本用法

无需身份验证即可连接到 AMQP 1.0.0 服务器。定义 AMQPOptions 属性值、虚拟主机，然后在 TsgcWebSocketClient 中设置服务器的 Host 和 Port。

如果您使用 TCP 普通连接，请将 TsgcWebSocketClient 属性 Specifications.RFC6455 设为 false。

```
// Creating AMQP client
oAMQP = new TsgcWSPClient_AMQP1(this);
// Creating WebSocket client
oClient = new TsgcWebSocketClient(this);
// Setting WebSocket specifications
oClient->Specifications->RFC6455 = false;
// Setting WebSocket client properties
oClient->Host = L"amqp_host_address";
oClient->Port = 5672;
// Assigning WebSocket client to AMQP client
oAMQP->Client = oClient;
// Activating WebSocket client
oClient->Active = true;
```

身份验证

若服务器需要身份验证，请使用 AMQP:Authentication 属性设置 Username/Password 的值，并将 AuthType 设置为 "amqp1authSASLPlain"。

```
// Creating AMQP client
oAMQP = new TsgcWSPClient_AMQP1(this);
// Setting AMQP authentication options
oAMQP->AMQPOptions->Authentication->AuthType = amqp1authSASLPlain;
oAMQP->AMQPOptions->Authentication->Username = L"sgc";
oAMQP->AMQPOptions->Authentication->Password = L"sgc";
// Creating WebSocket client
oClient = new TsgcWebSocketClient(this);
// Setting WebSocket specifications
oClient->Specifications->RFC6455 = false;
// Setting WebSocket client properties
oClient->Host = L"www.esegece.com";
oClient->Port = 5671;
oClient->TLS = true;
// Assigning WebSocket client to AMQP client
oAMQP->Client = oClient;
// Activating WebSocket client
oClient->Active = true;
```

连接 | 客户端 AMQP1 断开连接

客户端可以使用以下方法断开当前活动连接：

发送关闭原因

AMQP 客户端可通知服务器连接即将关闭，并提供关闭原因。使用 `Close` 方法向服务器请求关闭连接。

```
oAMQP.Close('invalid-frame', "The received frame has an invalid format.");
```

等待关闭

默认情况下，**Close** 方法是**异步的**，因此调用该方法后，代码将继续执行。如果您想**等待 Close 方法完成**并收到服务器发送的确认，请在 `Options` 参数中将 **Await** 设为 `True`。

```
void Close(const System::UnicodeString aCondition, const System::UnicodeString aDescription)
{
    TsgcAMQP1MethodOptions_Close *oOptions = new TsgcAMQP1MethodOptions_Close();
    try
    {
        oOptions->ErrorCondition = aCondition;
        oOptions->ErrorDescription = aDescription;
        oOptions->Await = true;
        AMQP1->Close(oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

关闭套接字连接

只需将 `TsgcWebSocketClient` 的 `Active` 属性设为 `False`。您可以阅读更多关于 [关闭连接](#) 的内容。

Connection | Idle Timeout

连接受到**空闲超时**阈值的约束。当在阈值时间内没有收到来自服务器的帧时，由客户端触发超时。
空闲超时以毫秒为单位，从收到最后一帧开始计算。
如果超过阈值，组件将向服务器发送关闭帧。如果服务器在 10 秒内没有响应，客户端将关闭 TCP 套接字。

空闲超时的值可以在以下属性中配置：

AMQPOptions.IdleTimeout

此属性中设置的值将在打开 AMQP 连接时发送给服务器。若该值大于零且小于 `MaxInt` 的一半，将启用内部计时器以检查空闲超时是否未超过。

示例：设置 `IdleTimeout` 值为 60 秒

```
AMQPOptions.IdleTimeout = 60000
```

连接 | 连接状态

AMQP 1.0.0 定义以下连接状态：

- **amqp1csUnknown**：初始状态。
- **amqp1csStart**：在此状态下，连接已存在，但尚未发送或接收任何数据。这是实现在执行套接字连接或套接字接受后立即所处的状态。
- **amqp1csHeaderReceived**：在此状态下，已从对等方收到连接头，但尚未发送连接头。
- **amqp1csHeaderSent**：在此状态下，连接头已发送给对端，但尚未收到连接头。
- **amqp1csHeaderExchanged**：在此状态下，连接标头已发送给对端，并已从对端接收到连接标头。
- **amqp1csOpenPipe**：在此状态下，连接头和 open 帧均已发送，但尚未收到任何数据。
- **amqp1csOpenClosePipe**：在此状态下，连接标头、open 帧、所有流水线连接流量以及 close 帧已发送，但尚未收到任何内容。
- **amqp1csOpenReceived**：在此状态下，连接标头已交换。已收到来自对端的 Open 帧，但尚未发送 Open 帧。
- **amqp1csOpenSent**：在此状态下，连接头已交换，Open 帧已发送给对端，但尚未收到对端的 Open 帧。
- **amqp1csClosePipe**：在此状态下，连接头部已交换。已发送 open 帧、任何流水线连接流量及 close 帧，但尚未从对端接收到 open 帧。
- **amqp1csOpened**：在此状态下，连接标头和 open 帧均已发送和接收。
- **amqp1csCloseReceived**：在此状态下，已收到关闭帧，表明对等方已发起 AMQP 关闭。不再预期收到更多帧；但仍可发送帧。
如果需要，实现方可在此时执行 TCP 半关闭以关闭连接的读取端。
- **amqp1csCloseSent**：在此状态下，已向对端发送关闭帧，向连接写入任何内容均为非法。
但可能仍有入站帧到来。如有需要，实现方可在此时执行 TCP 半关闭以关闭连接的写入端。
- **amqp1csDiscarding**：DISCARDING 状态是 CLOSE SENT 状态的一个变体，其中关闭由错误触发。
在此状态下，连接上任何传入帧都必须被静默丢弃，直到收到对等方的关闭帧。
- **amqp1csEnd**：在此状态下，任何端点都不得再向连接写入任何内容。
连接可以安全关闭并丢弃。

AMQP 客户端拥有 **ConnectionState** 属性，您可以通过它检查客户端组件当前处于哪种连接状态。

连接 | AMQP1 身份验证

该组件具有以下身份验证方法：

- **amqp1authNone**：连接到服务器时不使用任何身份验证方法。
- **amqp1authSASLAnonymous**：以匿名方式连接。
- **amqp1authSASLPlain**：默认方式，使用用户名/密码认证。
- **amqp1authSASLExternal**：目前不支持。

SASL 身份验证

最常见的身份验证方式是使用 **amqp1authSASLPlain** 类型。该验证类型可在 AMQP1 组件中通过访问 `AMQPOptions.Authentication` 属性来启用。

- **AuthType**：选择 `amqp1authSASLPlain`
- **Username**：用于 SASL 身份验证的用户名。
- **Password**：用于 SASL 认证的密钥值。

SASL 身份验证的结果可以通过事件 **OnAMQP1SASLAuthentication** 获取。

```
void __fastcall OnAMQP1SASLAuthentication(System::TObject* Sender, TsgcAMQP1SaslCode aCode,
    const System::UnicodeString aDescription, bool &Handled)
{
    ShowMessage("#sas1-authentication: " + aDescription);
}
```

连接 | Azure MessageBus

AMQP（高级消息队列协议）是一种强大的消息传递系统，旨在促进不同节点上各类容器之间的通信。它同时规范了消息传输协议和消息本身的结构框架，确保通信的一致性和可靠性。如需深入了解 AMQP 基础知识，请参阅我们的 AMQP 入门指南。

eSeGeCe 库中的 AMQP 组件支持与主流云消息代理（包括 Amazon MQ 和 Azure Service Bus）的无缝集成。本指南重点介绍如何使用 AMQP 组件连接 Azure Service Bus，演示如何构建能够高效发送和接收消息的多租户应用程序。

该组件提供了全面的实现，支持队列、主题和订阅等关键特性，是现代 IoT 和企业应用程序的理想选择。

Azure 配置

首先，在 Azure 门户中创建一个**服务总线资源**。资源建立后，请记录资源的**域名**，这对于集成和配置至关重要。

命名空间成功创建后，您可以直接从 Azure 门户的**命名空间概览**中管理和监控它。这个集中式界面提供对关键管理工具和设置的访问，支持对 Service Bus 资源进行无缝管理。

使用 **SAS 身份验证**时，用户名为 **SAS 策略名称**，密码为主密钥或辅助密钥。

```
// Create TCP client
TsgcWebSocketClient* oClient = new TsgcWebSocketClient(nullptr);
oClient->Specifications->RFC6455 = false;
oClient->Host = "esegece.servicebus.windows.net";
oClient->Port = 5671;
oClient->TLS = true;
// Create AMQP1 protocol client
TsgcWSCClient_AMQP1* oAMQP1 = new TsgcWSCClient_AMQP1(nullptr);
oAMQP1->Specifications->RFC6455 = false;
oAMQP1->AMQPOptions->Authentication->AuthType = amqp1authSASLPlain;
oAMQP1->AMQPOptions->Authentication->Username = "RootManageSharedAccessKey";
oAMQP1->AMQPOptions->Authentication->Password = "BhJ78+w8kMXhS/eE/nBy0cRzodx9tipbi+ASbAXIaH8=";
oAMQP1->Client = oClient;
// Connect to the server
oClient->Active = true;
```

Azure CBS 身份验证

Azure Service Bus 通过 AMQP 上的基于声明的安全（CBS）在初始 SASL 握手之后授权发送方和接收方。客户端在 **\$cbs** 节点打开管理链接，并发送一个包含共享访问签名（SAS）令牌或由 Microsoft Entra ID 颁发的 JSON Web Token（JWT）的 **put-token** 请求。代理验证令牌后，授权将在其有效期内缓存，应用程序随即可针对队列、主题或订阅创建发送和接收链接。

AMQP1 客户端通过两个辅助方法自动化此流程：

- **CreateAzureCbsSasToken** 建立一对 CBS 发送方/接收方链路，为目标实体生成 SAS 令牌，并将其发布到 **\$cbs**。使用共享访问策略进行身份验证时请调用此方法。
- **CreateAzureCbsJWT** 遵循相同的 CBS 交换流程，但在向 **\$cbs** 发送 JWT 之前，会使用客户端凭据授权从 Microsoft Entra ID（Azure AD）获取访问令牌。

两种方法都需要活动的 AMQP 连接，并接受以下参数：

- **aName**：内部创建的 CBS 链路对的标识符。
- **aNameSpace** 和 **aEntityName**：Service Bus 命名空间（不含 **.servicebus.windows.net** 后缀）以及用于构建令牌受众的队列、主题或订阅路径。
- **aKeyName/aKeyValue**：SAS 令牌的共享访问策略名称和密钥。组件对令牌进行签名，并使用令牌类型 **servicebus.windows.net:sastoken** 发送。

- **aTenant**、**aApplicationId**、**aSecret**：Microsoft Entra（Azure AD）目录 ID、应用程序（客户端）ID 以及客户端密钥，用于通过客户端凭据流请求 JWT。
- **aListeningPort**（JWT）：OAuth 2.0 重定向的本地 HTTP 端口（未提供时默认为 8080）。
- **aExpiration** 和 **aTimeout**：已签发令牌的生命周期（秒）以及 CBS 协商的最大等待时间（毫秒）。
- **aRaiseIfError**：设置为 *True* 时，如果令牌获取或 CBS 响应失败，该方法将引发异常。

以下示例说明了如何在发送消息之前使用 CBS 进行身份验证。

```
// Create TCP client
TsgcWebSocketClient* oClient = new TsgcWebSocketClient(nullptr);
oClient->Specifications->RFC6455 = false;
oClient->Host = "esegece.servicebus.windows.net";
oClient->Port = 5671;
oClient->TLS = true;
// Create AMQP1 protocol client
TsgcWScClient_AMQP1* oAMQP1 = new TsgcWScClient_AMQP1(nullptr);
oAMQP1->Specifications->RFC6455 = false;
oAMQP1->AMQPOptions->Authentication->AuthType = amqp1authSASLAnonymous;
oAMQP1->Client = oClient;
// Connect and publish SAS token through CBS
oClient->Active = true;
// wait the client is connected
oAMQP1->CreateAzureCbsSasToken("cbs", "esegece", "queue1",
    "RootManageSharedAccessKey",
    "Bhj78+w8kMXhS/eE/nBy0cRzodx9tipbi+ASbAXIaH8=",
    3600, 10000, true);
```

下一示例专注于使用 JWT 进行 Microsoft Entra ID（Azure AD）身份验证。它展示了如何使用客户端凭据流请求令牌，并在创建用于收发消息的链接之前将其发布到 **\$cbs**。

```
// Create TCP client
TsgcWebSocketClient* oClient = new TsgcWebSocketClient(nullptr);
oClient->Specifications->RFC6455 = false;
oClient->Host = "esegece.servicebus.windows.net";
oClient->Port = 5671;
oClient->TLS = true;
// Create AMQP1 protocol client
TsgcWScClient_AMQP1* oAMQP1 = new TsgcWScClient_AMQP1(nullptr);
oAMQP1->Specifications->RFC6455 = false;
oAMQP1->AMQPOptions->Authentication->AuthType = amqp1authSASLAnonymous;
oAMQP1->Client = oClient;
// Connect and publish JWT through CBS
oClient->Active = true;
oAMQP1->CreateAzureCbsJWT("cbs", "esegece", "queue1",
    "00000000-0000-0000-0000-000000000000", // Tenant ID
    "11111111-1111-1111-1111-111111111111", // Application ID
    "client-secret", 8080, 3600, 10000, true);
```

Commands | AMQP1 Sessions

在 AMQP（高级消息队列协议）1.0.0 规范的上下文中，会话表示客户端与消息代理之间通信的逻辑上下文。以下是 AMQP 1.0.0 会话的详细说明：

- **逻辑上下文**：会话在 AMQP 客户端（生产者或消费者）与 AMQP 代理之间建立用于消息操作的逻辑上下文。它提供了一种在单个连接中将相关消息操作归组的方式。
- **通信通道**：会话充当发送和接收消息的通信通道。它们封装消息交换、确认和流量控制机制。
- **事务边界**：会话定义消息操作的事务边界。它们允许将多个消息发送或接收分组为单个原子单元，确保会话内的所有操作要么全部成功处理，要么均不处理。
- **流量控制**：会话支持流量控制机制，以调节客户端与代理之间的消息交换速率。流量控制有助于防止任何一方的资源过载，确保高效可靠的消息传递。
- **生命周期管理**：会话具有从创建到关闭的生命周期。客户端可以在单个连接内建立多个会话，以实现消息处理的并行化或隔离消息流。
- **资源分配**：会话可与代理内的特定资源（如队列、主题或订阅）相关联。会话内发送或接收的消息与这些资源绑定，从而实现精准的消息路由和投递。

总而言之，AMQP 1.0.0 会话为 AMQP 客户端和代理之间的消息交换提供了逻辑上下文，有助于在消息系统中实现事务完整性、流控制和资源管理。它定义了执行消息操作的边界，有助于确保消息的高效可靠交换。

打开会话

方法 **CreateSession** 以给定名称创建新会话（若为空则创建随机名称会话），如果会话已存在则引发异常。客户端允许您使用同一 AMQP 连接创建多个会话。

会话成功创建后，将触发 **OnAMQPSessionOpen** 事件。

```
oAMQP->CreateSession("MySession");
oAMQP->OnAMQPSessionOpen = OnAMQPSessionOpenEvent;
void __fastcall TMyClass::OnAMQPSessionOpenEvent(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Frame *aFrame)
{
    ShowMessage("#session-open: " + aSession->Id);
}
```

CreateSession 方法返回包含会话信息的 **TsgcAMQP1Session** 类。

等待打开会话

默认情况下，**CreateSession** 方法是异步的，调用该方法后代码会继续执行。若希望等待 **CreateSession** 方法完成并接收服务器发送的确认，请在 **Options** 参数中将 **Await** 属性设置为 **True**。

```
void OpenSession(const System::UnicodeString aSession)
{
    TsgcAMQP1MethodOptions_SessionOpen *oOptions = new TsgcAMQP1MethodOptions_SessionOpen();
    try
    {
        oOptions->Await = true;
        AMQP1->CreateSession(aSession, oOptions);
    }
    finally
    {
        delete oOptions;
    }
}
```

关闭会话

要关闭现有会话，请使用方法 **CloseSession** 并传入要关闭的会话名称。

会话成功关闭后，将触发 **OnAMQPSessionClose** 事件。

```
oAMQP->CloseSession("MySession");
oAMQP->OnAMQPSessionClose = OnAMQPSessionCloseEvent;
// OnAMQPSessionCloseEvent implementation
void __fastcall TMyClass::OnAMQPSessionCloseEvent(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Frame *const aEnd, TsgcAMQP1Error *const aError)
{
    ShowMessage("#session-close: " + aSession->Id + " [" + IntToStr(aSession->Channel) + "] reason: " + aEnd->ErrorMessage);
}
```

等待关闭会话

默认情况下，**CloseSession** 方法是异步的，因此调用该方法后代码继续执行。如果您希望等待 **CloseSession** 方法完成并收到服务器发送的确认，请在 **Options** 参数中将属性 **Await** 设置为 **True**。

```
void CloseSession(const System::UnicodeString aSession)
{
    TsgcAMQP1MethodOptions_SessionClose *oOptions = new TsgcAMQP1MethodOptions_SessionClose();
    try
    {
        oOptions->Await = true;
        AMQP1->CloseSession(aSession, oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

命令 | AMQP1 链路

在 AMQP（高级消息队列协议）1.0.0 规范中，链路表示 AMQP 客户端与消息 broker 之间的单向通信信道。让我们深入了解 AMQP 1.0.0 链路的内容：

- **通信通道**：链路作为 AMQP 发送方和接收方之间消息流动的通道。它允许消息在一个方向上传输，既可以从发送方到接收方，也可以反向传输。
- **单向流**：每条链路均为单向，即消息只能沿链路的一个方向传输。若需要双向通信，必须建立两条链路，每个方向各一条。
- **消息传输**：消息根据 AMQP 协议规则在链接间传输。这些消息可包含有效载荷、消息属性及通信所需的附加元数据。
- **资源绑定**：链接与 AMQP 代理中的特定资源（如队列、主题或交换机）相关联。通过链接发送或接收的消息将定向到这些资源或来自这些资源。
- **流量控制**：链路支持流量控制机制，以调节消息发送或接收的速率。流量控制确保发送方和接收方均不会被交换的消息量所压垮。
- **生命周期管理**：链路具有生命周期，从建立开始，到关闭结束。可以根据需要动态创建，并在不再需要时关闭。
- **寻址**：链路由唯一地址标识，该地址指定了通信的源端点和目标端点。这些地址允许客户端和代理识别并建立与相应端点的连接。
- **事务边界**：链接定义消息操作的事务边界。它们支持将多个消息发送或接收操作分组为单个原子单元，从而确保消息传递的一致性和可靠性。

总而言之，AMQP 1.0.0 链路为 AMQP 客户端与代理之间的单向通信提供了手段，在消息传输系统内支持消息传输、流量控制、资源绑定、寻址和事务完整性。它们构成了 AMQP 协议中消息交换的基本构建块。

有 2 种链接类型：

- **发送方链接**：这些链接用于发送消息。
- **接收方链接**：这些链接用于接收消息。

每次创建或删除新链路时，都会触发以下事件：

- **OnAMQPLinkOpen**：当创建新链接时触发此事件。使用 `aLink.Mode` 属性检查链接是处于接收方还是发送方模式。
- **OnAMQPLinkClose**：当链接关闭时触发此事件。

命令 | AMQP1 发送方链路

在 AMQP 1.0.0 协议中，**发送方链接**是在 AMQP 客户端和 AMQP 服务器之间建立的**通信通道**，专用于**发送消息**。它在 AMQP 会话的上下文中运行，会话代表客户端与服务器之间通信的逻辑通道。

创建发送方链路

要创建新的发送方链接，请调用 **CreateSenderLink** 方法，该方法包含以下参数：

- **Session**：将附加发送链接的会话名称。
- **Name**：（可选）发送方链路的名称，若未设置，将自动分配随机名称。
- **Target**：（可选）您可以通过设置"target"参数指定远程主机上接收消息的目标。但在某些场景中，指定目标可能不是必需的。在这种情况下，提供空字符串即可。
- **SndSettleMode**：（默认混合）AMQP 通过消息结算机制来协商投递保证。建立链路时，发送方和接收方协商并商定结算模式（每个角色各一种）。发送方在以下模式之一下运行：
 - **amqp1ssmSettled**：消息发送后即被视为成功投递并已确认。
 - **amqp1ssmUnsettled**：消息在被接收方明确接受或拒绝之前不被视为已结算。这允许对消息处理和进行处理进行更多控制。
 - **amqp1ssmMixed**：在单个 AMQP 会话中可以混合使用已结算和未结算模式。使用 **SendMessage** 方法的 **MessageOptions** 参数来配置消息是否已结算。

当发送方链接成功创建后，将触发事件 **OnAMQPLinkOpen**。

```
oAMQP1->CreateSenderLink("MySession", "MySenderLink");
oAMQP1->OnAMQPLinkOpen = AMQP1AMQPLinkOpen;
void __fastcall TMyForm::AMQP1AMQPLinkOpen(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *const aLink)
{
    ShowMessage("#link-open: " + aLink->Name);
}
```

等待创建发送链路

默认情况下，**CreateSenderLink** 方法是**异步**的，调用该方法后代码将继续执行。若要**等待 CreateSenderLink 方法完成**并收到服务器发送的确认，请在 Options 参数中将 **Await** 属性设为 True。

```
void CreateSenderLink(const System::UnicodeString aSession, const System::UnicodeString aSender)
{
    TsgcAMQP1MethodOptions_CreateSenderLink *oOptions = new TsgcAMQP1MethodOptions_CreateSenderLink();
    try
    {
        oOptions->Await = true;
        AMQP1->CreateSenderLink(aSession, aSender, L"", oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

发送消息

要发送新消息，请调用方法 **SendMessage**，该方法包含以下参数：

- **Session**：会话名称。
- **Link**：发送方链接的名称。
- **Text**：字符串消息的文本内容。

```
oAMQP1->SendMessage("MySession", "MySenderLink", "My first AMQP Message");
```

混合模式发送消息

当发送方链接以混合模式（默认）创建时，发送消息时用户可以设置消息是否已**结算**或**未结算**。使用 **MessageOptions** 参数定义消息是否已结算。

```
TsgcAMQP1MessageOptions* oMessageOptions = new TsgcAMQP1MessageOptions();
try
{
    oMessageOptions->Settled = true;
    oAMQP1->SendMessage("MySession", "MySenderLink", "MyMessage", "message-id", oMessageOptions);
}
__finally
{
    delete oMessageOptions;
}
```

关闭发送方链路

要关闭现有发送方链路，请调用 **CloseLink** 方法，该方法包含以下参数：

- Session：包含该链路的会话名称。
- Link：发送方链路名称。
- 错误：（可选）您可在此处说明链接关闭的原因。

当发送链路成功关闭后，将触发 **OnAMQP1LinkClose** 事件。

```
oAMQP->CloseLink("MySession", "MySenderLink");
void __fastcall TForm1::OnAMQP1LinkCloseEvent(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *cc
{
    ShowMessage("#link-close: " + aLink->Name);
}
```

等待关闭发送链路

默认情况下，**CloseLink** 方法是**异步的**，因此调用该方法后代码会继续执行。如果您希望等待 **CloseLink** 方法完成并收到服务器发送的确认，请在 Options 参数中将 **Await** 属性设置为 True。

```
void CloseSenderLink(const System::UnicodeString aSession, const System::UnicodeString aSenderLink)
{
    TsgcAMQP1MethodOptions_CloseLink *oOptions = new TsgcAMQP1MethodOptions_CloseLink();
    try
    {
        oOptions->Await = true;
        AMQP1->CloseLink(aSession, aSenderLink, oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```


命令 | AMQP1 接收链路

在 AMQP 1.0.0 协议中，**接收链路**是在 AMQP 客户端和 AMQP 服务器之间建立的**通信信道**，用于**接收消息**。它在 AMQP 会话的上下文中运行，该会话表示客户端与服务器之间通信的逻辑信道。

创建接收方链路

要创建新的接收链路，请调用 **CreateReceiverLink** 方法，该方法包含以下参数：

- **Session**：将附加发送链路的会话名称。
- **Name**：（可选）发送方链路的名称，若未设置，将自动分配随机名称。
- **Source**：（可选）可以配置 Source 以指示远程主机上应充当发送方的节点的位置。在某些情况下，可能不需要指定此地址。在此类情况下，只需为参数提供空字符串即可。
- **ReadMode**：（默认为 **amqp1srmAuto**）接收方链路可以以两种模式之一接收消息：
 - **amqp1srmAuto**：自动模式，在此模式下，接收方主动确保消息可用后立即接收。每当新消息到达时，它自动监听并接收消息，无需任何明确指令。
 - **amqp1srmManual**：按需获取模式，在此模式下，接收方仅在被明确告知时才检索或获取新消息。与自动模式不同，接收方不会主动监听新消息，而是等待手动指令来获取下一条消息。
- **RcvSettleMode**：（默认为 **amqp1rsmFirst**）接收方链路在以下模式之一下运行：
 - **amqp1rsmFirst**：消息到达时，将立即处理并确认。如果消息在发送时尚未被确认，发送方将被告知消息已收到。
 - **amqp1rsmSecond**：到达的消息只有在发送方先确认后才会被确认。此外，当消息被接收时，如果消息在发送时尚未被确认，发送方将收到通知。

当接收方链路成功创建后，将触发 **OnAMQPLinkOpen** 事件。

```
oAMQP1->CreateReceiverLink("MySession", "MyReceiverLink");
oAMQP1->OnAMQPLinkOpen = AMQP1AMQPLinkOpen;
void __fastcall TMyForm::AMQP1AMQPLinkOpen(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *const aLink)
{
    ShowMessage("#link-open: " + aLink->Name);
}
```

等待创建接收链路

默认情况下，**CreateReceiverLink** 方法是**异步**的，调用该方法后代码会继续执行。如果您希望**等待 CreateReceiverLink 方法完成**并收到服务器发送的确认，请在 Options 参数中将 **Await** 设置为 **True**。

```
void CreateReceiverLink(const System::UnicodeString aSession, const System::UnicodeString aReceiver)
{
    TsgcAMQP1MethodOptions_CreateReceiverLink *oOptions = new TsgcAMQP1MethodOptions_CreateReceiverLink();
    try
    {
        oOptions->Await = true;
        AMQP1->CreateReceiverLink(aSession, aReceiver, L"", oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

同步消息

当接收方链路以手动 ReadMode 工作时，调用 **GetMessage** 方法获取新消息。该方法是同步的，即会等待直到超时（默认 10 秒）。调用该方法时，组件将信用额度增加一个单位，并等待新消息到达或超时。如果没有消息到达，信用额度将再次设置为零。

GetMessage 方法具有以下参数：

- **Session**：包含该链路的会话名称。
- **Link**：接收链路的名称。
- **Timeout**：（默认值 1000，即 10 秒）函数等待获取新消息的最长时间。

关闭接收链路

要关闭现有接收方链接，请调用 **CloseLink** 方法，该方法包含以下参数：

- **Session**：包含该链路的会话名称。
- **Link**：接收链路的名称。
- **错误**：（可选）此处可设置关闭链接的原因。

当接收方链接成功关闭时，将触发事件 **OnAMQPLinkClose**。

```
oAMQP1->CloseLink("MySession", "MyReceiverLink");
void __fastcall TForm1::OnAMQPLinkCloseEvent(TObject *Sender, TsgcAMQP1Session *const aSession, TsgcAMQP1Link *cc
{
    ShowMessage("#link-close: " + aLink->Name);
}
```

Await Close Receiver Link

默认情况下，**CloseLink** 方法是异步的，因此调用该方法后代码会继续执行。如果您希望等待 **CloseLink** 方法完成并收到服务器发送的确认，请在 Options 参数中将 **Await** 属性设置为 True。

```
void CloseReceiverLink(const System::UnicodeString aSession, const System::UnicodeString aReceiverLink)
{
    TsgcAMQP1MethodOptions_CloseLink *oOptions = new TsgcAMQP1MethodOptions_CloseLink();
    try
    {
        oOptions->Await = true;
        AMQP1->CloseLink(aSession, aReceiverLink, oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

AMQP1 | 发送消息

请先阅读 [AMQP1 发送链接](#)，了解如何创建发送链接。

发送消息

使用方法 **SendMessage**，传递会话和 SenderLink 名称，向 AMQP1 服务器发送文本消息。该方法具有以下参数：

- **Session**：会话名称。
- **Link**：发送方链接的名称。
- **Text**：消息的文本内容。
- **MessageId**：（可选）消息的 id，在使用未结算模式时可用于确认服务器是否已处理该消息。
- **Options**：（可选）允许在发送消息时自定义一些选项。
 - **Settled**：在混合模式的发送方链路中，发送消息时可自定义 Settled 属性。
 - **Await**：如果消息未结算且值为 true，代码将等待直到消息被服务器处理或超时。
 - **Timeout**：当 await 为 true 时，等待的毫秒数值（默认 10000）。
 - **RaiseTimeoutException**：如果超时，则抛出异常（默认为 true）。

```
oAMQP1->SendMessage("MySession", "MySenderLink", "My first AMQP Message");
```

等待发送消息

默认情况下，当以未结算方式发送消息时，**SendMessage** 方法是异步的；将属性 **Await** 设置为 true，客户端将等待直到收到服务器确认消息已处理。

```
void SendMessageAwait(const System::UnicodeString aSession, const System::UnicodeString aSenderLink, const System::UnicodeString aText, const System::TimeSpan timeout)
{
    TsgcAMQP1MethodOptions_SendMessageAck *oOptions = new TsgcAMQP1MethodOptions_SendMessageAck();
    try
    {
        oOptions->Settled = false;
        oOptions->Await = true;
        AMQP1->SendMessage(aSession, aSenderLink, aText, L"message-id", oOptions);
    }
    __finally
    {
        delete oOptions;
    }
}
```

事件

发送消息时，有 2 个事件可用于了解消息何时发送以及消息是否已被服务器处理（发送未确认消息时）。

- **OnAMQPMessageSent**：消息发送到服务器后调用此事件。调用 **SendMessage** 方法时，消息存储在内部队列中并由辅助线程处理，消息发送后将触发此事件。
- **OnAMQPMessageSentAck**：当客户端收到消息已被 AMQP1 服务器处理的确认时调用此事件。

```
void __fastcall OnAMQPMessageSentAck(System::TObject* Sender, TsgcAMQP1Session* const aSession, TsgcAMQP1SenderLink* const aLink, const System::UnicodeString aMessageId,
```

```
TsgcAMQP1FrameDeliveryStates aDeliveryState, TsgcAMQP1FrameDisposition aDisposition)
{
    System::UnicodeString vMessageId = aMessageId;
    switch (aDeliveryState)
    {
        case amqp1fdtsAccepted:
            ShowMessage("#msg-accepted: " + vMessageId);
            break;
        case amqp1fdtsRejected:
            ShowMessage("#msg-rejected: " + vMessageId + " " +
                TsgcAMQP1FrameRejected(aDisposition.State)->Error->Condition + " " +
                TsgcAMQP1FrameRejected(aDisposition.State)->Error->Description);
            break;
        case amqp1fdtsReleased:
            ShowMessage("#msg-released: " + vMessageId);
            break;
        case amqp1fdtsModified:
            ShowMessage("#msg-modified: " + vMessageId + " " +
                TsgcAMQP1FrameModified(aDisposition.State)->MessageAnnotations);
            break;
        case amqp1fdtsReceived:
            ShowMessage("#msg-received: " + vMessageId);
            break;
    }
}
```

AMQP1 | 读取消息

每次收到新消息时，都会触发 **OnAMQPMessage** 事件。

TsgcAMQP1Message 实例包含收到的消息。您可以使用属性 **aMessage.ApplicationData.AMQPValue.Value** 访问文本消息。

要指定交付结果，请使用 **DeliveryState** 参数。默认情况下，所有消息都处于已接受状态，但您可以设置以下之一：

- **amqp1mdtsAccepted**：消息已成功处理。
- **amqp1mdtsRejected**：消息处理失败。使用属性 **DeliveryState.Rejected** 设置错误。
- **amqp1mdtsReleased**：该消息未被处理且不会被处理。
- **amqp1mdtsModified**：与 **amqp1mdtsReleased** 相同，但您可以使用 **DeliveryState.Modified** 属性添加额外数据。

```
void __fastcall OnAMQPMessage(System::TObject* Sender, TsgcAMQP1Session* const aSession,
    TsgcAMQP1ReceiverLink* const aLink, TsgcAMQP1Message* const aMessage,
    TsgcAMQP1MessageDeliveryState &DeliveryState)
{
    if (aMessage->ApplicationData->AMQPValue->Value == "xxx")
    {
        DeliveryState.State = amqp1mdtsRejected;
        DeliveryState.Rejected->Error->Condition = "amqp-error-processing";
        DeliveryState.Rejected->Error->Description = "Value received was not the expected.";
    }
    else
    {
        DeliveryState.State = amqp1mdtsAccepted;
    }
}
```

Protocol STOMP

STOMP 是简单（或流式）文本定向消息协议。STOMP 提供了一种互操作的线路格式，使 STOMP 客户端能够与任何 STOMP 消息代理通信，从而在多种语言、平台和代理之间实现便捷广泛的消息互操作性。

我们的 STOMP 客户端组件支持以下 STOMP 版本：1.0、1.1 和 1.2。

组件

TsgcWSPClient_STOMP：通用 STOMP 协议客户端，允许您连接到任何 STOMP 服务器。

TsgcWSPClient_STOMP_RabbitMQ：RabbitMQ 代理的 STOMP 客户端。

TsgcWSPClient_STOMP_ActiveMQ：适用于 ActiveMQ Broker 的 STOMP 客户端。

TsgcWSPClient_STOMP

这是客户端协议 STOMP 组件。

简介

这是客户端协议 STOMP 组件。您需要将此组件拖放到窗体上，并使用 Client 属性选择一个 [TsgcWebSocketClient](#) 组件。

参考

- 属性 — 组件上有 8 个属性可用。
- 方法 — 11 个公共方法。
- 事件 — 6 个可处理的事件。

TsgcWSPClient_STOMP_RabbitMQ

这是适用于 RabbitMQ Broker 的客户端 STOMP 协议组件。

简介

这是 RabbitMQ 代理的客户端协议 STOMP 组件。您需要将此组件放置在表单上，并使用 Client 属性选择一个 [TsgcWebSocketClient](#) 组件。

目的地

STOMP 规范未规定代理必须支持哪些类型的目的地，而是将 SEND 和 MESSAGE 帧中 destination 头的值留给代理自行定义。RabbitMQ STOMP 适配器支持多种不同的目的地类型：

- **主题**：向瞬态和持久主题 SEND 和 SUBSCRIBE。
- **Queue**：向 STOMP 网关管理的队列执行 SEND 和 SUBSCRIBE 操作。
- **QueueOutside**：向在 STOMP 网关外部创建的队列发送消息（SEND）并进行订阅（SUBSCRIBE）。
- **TemporaryQueue**：创建临时队列（仅在 reply-to 头中使用）。
- **Exchange**：向任意路由键发送消息（SEND），并订阅任意绑定模式（SUBSCRIBE）。

参考

- Properties — 组件上共有 8 个可用属性。
- 方法 — 25 个公共方法。
- 事件 — 6 个可处理的事件。

TsgcWSPClient_STOMP_ActiveMQ

这是 ActiveMQ Broker 的客户端协议 STOMP 组件。

简介

这是 ActiveMQ Broker 的客户端协议 STOMP 组件。您需要将此组件放置在表单上，并使用 Client 属性选择一个 [TsgcWebSocketClient](#) 组件。

目的地

STOMP 规范没有规定 broker 必须支持哪些目的地类型，SEND 和 MESSAGE 帧中目的地头的值取决于具体的 broker。Active STOMP 适配器支持多种不同的目的地类型：

- **主题**：向瞬态和持久主题 SEND 和 SUBSCRIBE。
- **Queue**：向 STOMP 网关管理的队列执行 SEND 和 SUBSCRIBE 操作。

发布选项

请注意，STOMP 的设计尽可能简单，因此任何脚本语言/平台都能以最小的工作量与其他平台交换消息。STOMP 允许在每个请求上使用可插拔的请求头，例如发送和接收消息。ActiveMQ 对 STOMP 协议有若干扩展，以使 Stomp 客户端能够支持 JMS 语义。OpenWire JMS 生产者可以向 Stomp 消费者发送消息，Stomp 生产者也可以向 OpenWire JMS 消费者发送消息。Stomp 到 Stomp 的配置可以使用更丰富的 JMS 消息控制。

STOMP 支持 SENT 消息上的以下标准 JMS 属性：

- **CorrelationId**：良好的消费者实现应在其发送的响应中添加此标头。
- **Expires**：消息的过期时间。
- **JMSXGroupID**：指定消息组。
- **JMSXGroupSeq**：可选头部，指定消息组中的序列号。
- **Persistent**：消息是否持久化。
- **Priority**：消息优先级。
- **ReplyTo**：您应发送回复的目标地址。
- **MsgType**：消息类型。

参考

- 属性 — 组件上可用的 9 个属性。
- 方法 — 16 个公共方法。
- 事件 — 6 个可处理的事件。

协议 AppRTC

WebRTC（Web 实时通信）是由 W3C 起草的 API 定义，旨在实现无需插件的浏览器到浏览器语音通话、视频聊天和 P2P 文件共享应用。作为一套标准，WebRTC 为任何浏览器提供了在无需安装插件或第三方软件的情况下共享应用数据和进行电话会议的能力。

WebRTC 组件通过 JavaScript API 访问。目前正在开发的有网络流 API（表示音频或视频数据流）以及 PeerConnection API（允许两个或多个用户进行浏览器间通信）。此外还在开发 DataChannel API，支持实时游戏、文本聊天、文件传输等其他类型数据的通信。

[appr.tc](#) 是由 Google 和 Mozilla 开发的 WebRTC 演示应用程序，它使两个浏览器能够使用 WebRTC API 相互"通话"。

组件

[TsgcWSPServer_AppRTC](#)：服务器协议 AppRTC VCL 组件。

TsgcWSPServer_AppRTC

这是服务器协议 AppRTC 组件。

简介

这是服务器协议 AppRTC 组件。您需要将此组件放置在窗体上，并使用 Server 属性选择 [TsgcWebSocketServer](#) 组件。

参数

- **IceServers**：在此处配置 WebRTC 连接的 TURN/STUN 服务器。
- **RoomLink**：访问房间的基础 URL。示例：<https://mydemo.com/r/>
- **WebSocketURL**：WebSocket 服务器的 URL。示例：<wss://mydemo.com>

WebRTC 协议需要 STUN/TURN 服务器，示例使用公共 STUN/TURN 服务器进行测试。如需部署到生产系统，需要使用专用的 STUN/TURN 服务器。

注册用户可以下载 **Windows 版 Coturn 服务器** 的编译二进制文件。了解更多关于 [COTURN STUN/TURN](#) 的信息。

IceServers 配置

如果您的 STUN/TURN 服务器运行在以下 IP 地址：51.122.4.88，监听端口 3478，连接用户为 "apprtc"，凭据为 "secret"，请按如下方式配置 IceServers：

```
{
  "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"
}
```

参考

- 属性 — 组件上提供 5 个属性。
- 方法 — 0 个公开方法。
- 事件 — 共 11 个可处理的事件。

协议 WebRTC

WebRTC（Web 实时通信）是由 W3C 起草的 API 定义，旨在实现无需插件的浏览器到浏览器语音通话、视频聊天和 P2P 文件共享应用。作为一套标准，WebRTC 为任何浏览器提供了在无需安装插件或第三方软件的情况下共享应用数据和进行电话会议的能力。

WebRTC 组件通过 JavaScript API 访问。目前正在开发的有网络流 API（表示音频或视频数据流）以及 PeerConnection API（允许两个或多个用户进行浏览器间通信）。此外还在开发 DataChannel API，支持实时游戏、文本聊天、文件传输等其他类型数据的通信。

组件

[TsgcWSPServer_WebRTC](#)：服务器协议 WebRTC VCL 组件。

参数

- **IceServers**：在此处可以为 WebRTC 连接配置 TURN/STUN 服务器。默认使用以下公共 STUN 服务器

```
{"iceServers": [{"url": "stun:stun.l.google.com:19302"}]}
```

浏览器测试

如果您想用您最喜欢的 Web 浏览器测试此协议，请输入以下 URL（您需要定义您的自定义主机和端口）

<http://host:port/webrtc.esegece.com.html>

TsgcWSPServer_WebRTC

这是服务器协议 WebRTC 组件。

简介

这是服务器协议 WebRTC 组件。您需要将此组件拖放到窗体中，并通过 Server 属性选择一个 [TsgcWebSocketServer](#) 组件。

WebRTC 协议需要 STUN/TURN 服务器，示例使用公共 STUN/TURN 服务器进行测试。如需部署到生产系统，需要使用专用的 STUN/TURN 服务器。

注册用户可以下载 **Windows 版 Coturn 服务器** 的编译二进制文件。了解更多信息关于 [COTURN STUN/TURN](#) 的信息。

参考

- 属性 — 组件上提供 5 个属性。
- 方法 — 2 个公共方法。
- 事件 — 11 个可处理的事件。

协议 WebRTC JavaScript

您可以在此处找到可用的方法，需要根据实际情况替换 `{%host%}` 和 `{%port%}` 变量，例如：如果您将 `sgcWebSocket` 服务器配置为在 `www.example.com` 网站的 80 端口监听，则需要配置：

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/webrtc.esegece.com.js"></script>
```

打开连接

当 `WebSocket` 连接打开时，浏览器会请求访问本地摄像头和麦克风，您需要允许访问。

```
<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>
```

打开 WebRTC 频道

当浏览器可访问本地摄像头和麦克风时，将触发 `"sgcmediastart"` 事件，之后即可尝试使用 `webrtc_connect` 过程连接到另一个客户端。

```
<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>
```

关闭 WebRTC 频道

```
<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>
```

协议 WAMP

WAMP 是一种开放的 WebSocket 子协议，提供两种异步消息模式：RPC 和 PubSub。

从技术上讲，WAMP 是一种官方注册的 WebSocket 子协议（运行在 WebSocket 之上），使用 JSON 作为消息序列化格式。

什么是 RPC？

远程过程调用（RPC）是一种消息传递模式，涉及两个角色的对等方：客户端和服务端。

服务器在知名端点下提供可调用的方法或过程。

客户端通过提供方法或过程端点以及调用的任何参数来调用远程方法或过程。

服务器将使用提供给调用的参数执行该方法或过程，并将调用结果返回给客户端。

什么是 PubSub？

发布与订阅 (PubSub) 是一种涉及三种角色（发布者、订阅者和代理）的消息传递模式。

发布者通过提供主题（即频道）作为抽象地址（而非特定对端）来发送（发布）事件。

订阅者首先提供其感兴趣的主体（又称频道）来接收事件。之后，订阅者将接收发布到该主题的任何事件。

broker 位于发布者和订阅者之间，对消息发布进行中介。broker 会维护每个主题的订阅者列表，以便将新发布的事件分发给相应的订阅者。

代理也可以自行触发事件，例如当代理同时作为 RPC 服务器时，在服务器上执行的方法可能触发 PubSub 事件。

总而言之，PubSub 通过中间人（即代理）将发布者和接收者解耦。

组件

[TsgcWSPServer_WAMP](#)：服务器协议 WAMP VCL 组件。

[TsgcWSPClient_WAMP](#)：客户端协议 WAMP VCL 组件。

[Javascript 组件](#)：客户端 Javascript 参考。

最常见的用途

- **RPC**
 - [简单 RPC](#)
 - [RPC 渐进式结果](#)
- **PubSub**
 - [订阅者](#)
 - [发布者](#)

浏览器测试

如果您想用您喜欢的 Web 浏览器测试此协议，请输入此 URL（您需要定义自己的主机和端口）

`http://host:port/wamp.esegce.com.html`

TsgcWSPServer_WAMP

这是服务器协议 WAMP 组件。

简介

这是服务器协议 WAMP 组件。您需要将此组件放置在窗体上，并使用 Server 属性选择一个 [TsgcWebSocketServer](#) 组件。

参考

- 属性 — 组件上有 4 个可用属性。
- 方法 — 4 个公共方法。
- Events — 14 个可处理的事件。

TsgcWSPClient_WAMP

这是客户端协议 WAMP 组件。

简介

这是客户端协议 WAMP 组件。您需要将此组件拖放到窗体上，并使用 Client 属性选择一个 [TsgcWebSocketClient](#) 组件。

参考

- 属性 — 组件上有 3 个可用属性。
- Methods — 7 个公有方法。
- 事件 — 13 个可处理的事件。

协议 WAMP JavaScript

您可以在此处找到可用的方法，需要根据实际情况替换 `{%host%}` 和 `{%port%}` 变量，例如：如果您将 sgcWebSocket 服务器配置为在 `www.example.com` 网站的 80 端口监听，则需要配置：

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/wamp.esegece.com.js"></script>
```

打开连接

```
<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>
```

发送新前缀

```
<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>
```

请求 RPC（远程过程调用）

```
<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>
```

订阅 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>
```

取消订阅 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>
```

发布消息

```
<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>
```

收到消息时显示提醒

```
<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>
```

在 OnCallResult 或 OnCallError 时显示警告

```
<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>
```

OnEvent 时显示提醒

```
<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>
```

在 OnConnect、OnDisconnect 和 OnError 事件时显示警告

```
<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>
```

关闭连接

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  socket.close();
</script>
```

获取连接状态

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  socket.state();
</script>
```

WAMP | 订阅者

订阅者首先提供其感兴趣的主题（又称频道）来接收事件。之后，订阅者将接收发布到该主题的任何事件。要从主题接收事件，订阅者首先必须订阅该主题。

WAMP 客户端

```
void OnMessageEvent(TsgcWSConnection *Connection, string Text)
{
    ShowMessage(Text);
}

oClient = new TsgcWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP->Client = oClient;
oClientWAMP->OnMessage = OnMessageEvent;
oClient->Active = true;

// Subscribe to topic after successful connect
oClient->Subscribe("myTopic");
```

WAMP 服务器

```
void OnSubscriptionEvent(TsgcWSConnection *Connection, string Subscription)
{
    ShowMessage("Subscribed: " + Subscription);
}

oServer = new TsgcWebSocketServer();
oServer->Port = 80;
oServerWAMP = new TsgcWSPServer_WAMP();
oServerWAMP->OnSubscription = OnSubscriptionEvent;
oServerWAMP->Server = oServer;
oServerWAMP->Active = true;
```

WAMP | 发布者

发布者通过提供主题（即频道）作为抽象地址而非具体对端来发布事件。只需调用 Publish 方法，并将主题名称和要发送的消息作为参数传入，该消息将被投递给该主题的所有订阅者。请注意，向某主题发布消息无需事先订阅该主题。

无需在服务器端配置任何内容，因为当收到发布消息时，消息会自动广播给客户端。

WAMP 客户端

```
oClient = new TsgcWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP->Client = oClient;
oClientWAMP->OnMessage = OnMessageEvent;
oClient->Active = true;

// Publish a message to all subscribers
oClient->Publish("myTopic", "Hello subscribers myTopic");
```

WAMP | 简单 RPC

WAMP 组件最常见的用途是客户端从服务器请求方法，服务器向客户端发送响应。客户端只需发送方法名称，并/或传递服务器计算结果所需的一些参数。服务器处理请求，如果成功，则向客户端发送包含结果的响应；如果发生任何错误，服务器向客户端发送错误响应。如您所见，只有一个请求和一个响应（成功或失败）。

示例：服务器有一个名为 **GetTime** 的方法，每次客户端请求此方法时，服务器返回服务器时间。

WAMP 服务器

```
void OnServerCall(TsgcWSConnection *Connection, const string CallId, const string ProcUri, const string Arguments)
{
    if (ProcUri == "GetTime")
    {
        oServerWAMP->CallResult(CallId, FormatDateTime("yyyyMMdd hh:nn:ss", Now));
    }
    else
    {
        oServer->WAMP->CallError(CallId, "Unknown method");
    }
}

oServer = new TsgcWebSocketServer();
oServer->Port = 80;
oServerWAMP = new TsgcWSPServer_WAMP();
oServerWAMP->OnCall = OnServerCallEvent();
oServerWAMP->Server = oServer;
oServer->Active = true;
```

WAMP 客户端

```
void OnCallResultClient(TsgcWSConnection *Connection, string CallId, string Result);
{
    ShowMessage(Result);
}

void OnCallErrorClient(TsgcWSConnection *Connection, string Error)
{
    ShowMessage(Error);
}

oClient = new TsgcWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP->OnCallResult = OnCallResultClient;
oClientWAMP->OnCallError = OnCallErrorClient;
oClientWAMP->Client = oClient;
oClient->Active = true;
// After client has connected, request GetTime from server
oClientWAMP->Call("GetTime");
```

WAMP | RPC 渐进结果

有时，远程过程调用需要多个结果才能完成请求，默认情况下 WAMP 1.0 协议不允许在调用中使用部分结果，这是 sgcWebSockets 库独有的功能。

该流程与简单的 RPC 非常相似，但在 CallResult 被调用以完成流程之前，会有 1 个或多个部分结果先行返回。基本上，客户端向服务器请求一个过程，服务器可以发送结果或错误。如果发送结果，可以是最终结果，也可以是还需发送更多结果。如果是最终结果，将调用方法 **CallResult**，进程随之结束。如果还有更多结果需要发送，将调用方法 **CallProgressResult**。

示例：客户端请求服务器提供一个每秒返回服务器时间的方法，并在 20 条消息后停止。

WAMP 服务器

```
void OnServerCall(TsgcWSConnection *Connection, const string CallId, const string ProcUri, const string Arguments)
{
    if (ProcUri == "GetProgressiveTime")
    {
        int vNum = StrToInt(Arguments);
        for (int i = 1; i = vNum; i++)
        {
            if (i == 20)
            {
                oServerWAMP->CallResult(CallId, FormatDateTime("yyyyMMdd hh:nn:ss", Now));
            }
            else
            {
                oServerWAMP->CallProgressiveResult(CallId, FormatDateTime("yyyyMMdd hh:nn:ss", Now));
            }
        }
    }
    else
    {
        oServer->WAMP->CallError(CallId, "Unknown method");
    }
}

oServer = new TsgcWebSocketServer();
oServer->Port = 80;
oServerWAMP = new TsgcWSPServer_WAMP();
oServerWAMP->OnCall = OnServerCallEvent();
oServerWAMP->Server = oServer;
oServer->Active = true;
```

WAMP 客户端

```
void OnCallResultClient(TsgcWSConnection *Connection, string CallId, string Result);
{
    ShowMessage(Result);
}

void OnCallProgressResultClient(TsgcWSConnection *Connection, string CallId, string Result);
{
    ShowMessage(Result);
}

void OnCallErrorClient(TsgcWSConnection *Connection, string Error)
{
    ShowMessage(Error);
}

oClient = new TsgcWebSocketClient();
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP->OnCallResult = OnCallResultClient;
oClientWAMP->OnCallProgressResult = OnCallProgressResultClient;
oClientWAMP->OnCallError = OnCallErrorClient;
oClientWAMP->Client = oClient;
oClient->Active = true;
```



```
// After client has connected, request GetTime from server  
oClientWAMP->Call("GetProgressTime");
```

协议 WAMP 2

WAMP 提供了一种基于开放 WebSocket 协议的统一应用路由方案，可与不同编程语言协同工作。

使用 WAMP 可以构建由松散耦合并以（软）实时方式通信的应用组件组成的分布式系统。

WAMP 的核心为应用程序组件之间的通信提供了两种模式：

- 发布与订阅（PubSub）
- 远程过程调用（RPC）

WAMP 易于使用、易于实现，并基于现代 Web 标准：WebSocket、JSON 和 URI。

组件

[TsgcWSPClient_WAMP2](#)：客户端协议 WAMP2 VCL 组件。

TsgcWSPClient_WAMP2

这是 Client Protocol WAMP2 组件。

简介

这是客户端协议 WAMP2 组件。您需要将此组件拖放到窗体上，并使用 Client 属性选择一个 [TsgcWebSocketClient](#) 组件。

会话方法

- **ABORT**：路由器和客户端均可通过发送 ABORT 消息来中止 WAMP 会话的建立。
Reason 必须是 URI。
Details 必须是一个字典，用于提供附加的、可选的关闭信息（见下文）。
 不期望对 ABORT 消息有任何响应。
- **GOODBYE**：WAMP 会话的生命周期始于路由器向客户端发送 WELCOME 消息，终于底层传输消失或一方发送 GOODBYE 消息后另一方也发送 GOODBYE 消息作为响应，从而显式关闭 WAMP 会话。
Reason 必须是 URI。
Details 必须是一个字典，允许提供额外的可选关闭信息。

发布/订阅方法

- **PUBLISH**：当发布者请求向某个主题发布事件时，它向 Broker 发送 PUBLISH 消息：
Request 是发布者选择的随机临时 ID，用于将 Broker 的响应与请求关联起来。
Options 是一个字典，允许您以可扩展的方式提供额外的发布请求详情。以下将进一步说明。
Topic 是发布到的主题。
Arguments 是应用层事件负载元素的列表，长度可以为零。
ArgumentsKw 是一个可选字典，包含以关键字参数形式提供的应用级事件负载。该字典可以为空。
 如果代理能够完成并允许发布，代理将把事件发送给已发布事件主题的所有当前订阅者。默认情况下，发布操作不需要确认，代理不会响应发布是否成功。
- **SUBSCRIBE**：订阅者通过向代理发送 SUBSCRIBE 消息来表明其对某个主题的订阅意向：
Request 必须是订阅者选择的随机临时 ID，用于将代理的响应与请求进行关联。
Options 必须是一个字典，以可扩展的方式提供额外的订阅请求详情。
Topic 是订阅者希望订阅的主题，必须是一个 URI。
- **UNSUBSCRIBE**：当订阅者不再希望接收某个订阅的事件时，将发送 UNSUBSCRIBE 消息
Request 必须是订阅者选择的随机临时 ID，用于将代理的响应与请求进行关联。
SUBSCRIBED.Subscription 必须是取消订阅所用的订阅 ID，该 ID 最初由代理分配给订阅者。

RPC 方法

- **CALL**：当调用方（Caller）希望调用远程过程时，将向 Dealer 发送 CALL 消息：
Request 是由调用方选择的随机临时 ID，用于将经销商的响应与请求关联。
Options 是一个字典，允许您以可扩展的方式提供额外的调用请求详情。
Procedure 是要调用的过程的 URI。

Arguments 是按位置排列的调用参数列表（每个参数类型任意）。该列表长度可以为零。
ArgumentsKw 是关键字调用参数的字典（每个参数为任意类型）。该字典可以为空。

- **REGISTERCALL**：被调用方通过向 Dealer 发送 REGISTER 消息来声明实现某个过程的端点可用：

Request 是由被调用方（Callee）随机生成的临时 ID，用于将 Dealer 的响应与请求进行关联。
Options 是一个字典，允许以可扩展方式提供额外的注册请求详情。下文将进一步描述。
Procedure 是被调用方希望注册的过程
- **UNREGISTERCALL**：当被调用方不再愿意提供已注册过程的实现时，它向 Dealer 发送 UNREGISTER 消息：

Request 是由被调用方（Callee）随机生成的临时 ID，用于将 Dealer 的响应与请求进行关联。
REGISTERED.Registration 是要撤销的注册 ID，该 ID 最初由 Dealer 分配给 Callee。
- **INVOCATION**：如果经销商能够履行（调解）调用并允许该调用，它会向实现该过程的相应被调用方发送 INVOCATION 消息：

Request 是经销商选择的随机临时 ID，用于将被调用者的响应与请求关联起来。
REGISTERED.Registration 是该过程在经销商处注册时所使用的注册 ID。
Details 是一个字典，允许您以可扩展的方式提供额外的调用请求详情。
CALL.Arguments 是调用方提供的原始位置调用参数列表。
CALL.ArgumentsKw 是调用者提供的原始关键字调用参数字典。
- **YIELD**：如果被调用方能够成功处理并完成调用的执行，则向 Dealer 发送 YIELD 消息作为回复：

INVOCATION.Request 是原始调用请求的 ID。
Options 是一个允许提供附加选项的字典。
Arguments 是位置结果元素的列表（每个元素为任意类型）。该列表长度可以为零。
ArgumentsKw 是关键字结果元素的字典（每个元素为任意类型）。字典可以为空。

参考

- 属性 — 该组件提供 3 个属性。
- 方法 — 11 个公共方法。
- 事件 — 可处理 20 个事件。

协议默认

这是使用 "JSONRPC 2.0" 消息实现的默认子协议，每次使用该协议发送消息时，都会创建一个具有以下属性的 JSON 对象：

jsonrpc：一个字符串，用于指定 JSON-RPC 协议的版本。必须严格为 "2.0"。

method：包含要调用的方法名称的字符串。以单词 rpc 后跟句点字符（U+002E 或 ASCII 46）开头的方法名称保留给 rpc 内部方法和扩展，不得用于其他用途。

params：结构化值，包含方法调用期间使用的参数值。此成员可以省略。

id：由客户端建立的标识符，如果包含则必须包含字符串、数字或 NULL 值。如果未包含，则假定为通知。该值通常不应为 Null [1]，数字不应包含小数部分 [2]

JSON 对象示例：

```
{"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
```

功能特性

- 发布/订阅消息模式，提供一对多的消息分发和应用程序解耦。支持通配符字符，因此您可以订阅频道层级结构。示例：如果您想订阅所有以 'news' 开头的频道，请调用 `Subscribe('news*')`。
- 一种与载荷内容无关的消息传输协议
- 确认已发送的消息。
- 支持事务性消息，通过服务器本地事务实现。当客户端提交事务时，服务器处理所有排队的消息。如果客户端回滚事务，则所有消息均被删除。
- 实现 QoS（服务质量）用于消息传递。

组件

[TsgcWSPClient_sgc](#)：服务器协议默认 VCL 组件。

[TsgcWSPClient_sgc](#)：客户端协议默认 VCL 组件。

[Javascript 组件](#)：客户端 Javascript 参考。

浏览器测试

如果您想用您喜欢的 Web 浏览器测试此协议，请输入此 URL（您需要定义自己的主机和端口）

`http://host:port/esegece.com.html`

TsgcWSPServer_sgc

这是服务器 Protocol Default 组件。

简介

这是服务器协议默认组件。您需要将此组件放置在表单上，并使用 Server 属性选择一个 [TsgcWebSocketServer](#) 组件。

参考

- 属性——组件上提供 6 个属性。
- 方法 — 6 个公共方法。
- 事件 — 16 个可处理的事件。

TsgcWSPClient_sgc

这是 *Client Protocol Default* 组件。

简介

这是默认的客户端协议组件。您需要在表单中放置此组件，并使用 Client 属性选择一个 [TsgcWebSocketClient](#) 组件。

参考

- 属性 — 组件上可用的 5 个属性。
- Methods — 11 个公共方法。
- 事件 — 15 个可处理的事件。

TsgclWWSPClient_sgc

这是 IntraWeb 客户端协议默认组件。

简介

这是 IntraWeb 客户端协议默认组件。您需要将此组件放置到窗体上，并使用 Client 属性选择一个 [TsgclWWebSocketClient](#) 组件。

参考

- Properties — 组件上共有 2 个可用属性。
- 方法 — 3 个公共方法。
- 事件 — 6 个可处理的事件。

协议默认 Javascript

默认协议 JavaScript，sgcWebSockets 使用 **sgcWebSocket.js** 和 **esegece.com.js** 文件。

您可以在此处找到可用的方法，需要根据实际情况替换 `{%host%}` 和 `{%port%}` 变量，例如：如果您将 sgcWebSocket 服务器配置为在 `www.example.com` 网站的 80 端口监听，则需要配置：

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/esegece.com.js"></script>
```

打开连接

```
<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>
```

发送消息

```
<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>
```

收到消息时显示提醒

```
<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>
```

向测试频道发布消息

```
<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>
```

显示收到事件消息的提示

```
<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>
```

调用 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>
```

处理 RPC 响应

```
<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>
```

调用通知

```
<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>
```

在事务中发送消息

```
<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>
```

订阅或取消订阅频道时显示警告

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
var socket = new sgcs('ws://{%host%}:{%port%}');
socket.on('sgcsubscribe', function(event)
{
    alert('subscribed: ' + event.channel);
});
socket.on('sgcunsubscribe', function(event)
{
    alert('unsubscribed: ' + event.channel);
});
</script>
```

在 OnConnect、OnDisconnect 和 OnError 事件时显示警告

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
var socket = new sgcs('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>
```

获取会话

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
var socket = new sgcs('ws://{%host%}:{%port%}');
socket.on('sgcsession', function(event)
{
    alert(event.guid);
});
socket.getSession();
</script>
```

关闭连接

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
socket.close();
</script>
```

获取连接状态

```
<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>
```

设置 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>
```

设置队列级别

```
<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>
```

数据集协议

此协议继承自默认协议，适用于您希望通过连接到此协议的客户端广播数据集变更的场景。它可以在 2 种模式下使用：

- 1. 复制数据库：**数据库变更将复制到所有客户端数据库，例如：服务器有一个包含股票报价的数据库，所有已连接的客户端接收报价变更。存在单个数据库（在服务器上），每个客户端有其自己的数据库。每次更新报价时，此变更将广播到所有已连接的客户端，每个客户端更新其自己的记录数据库。此模式类型使用 UpdateMode: upWhereAll 或 upWhereChanged。
- 2. 数据库更新：**此模式下，服务器和客户端共用一个数据库，每当有客户端更新数据库中的记录时，所有其他客户端都希望收到关于此更新的通知。对此模式请使用 UpdateMode: upRefreshAll。

最常见的用途

- 更新模式
 - [如何复制表](#)
 - [通知更新的工作方式](#)

它使用"JSON-RPC 2.0"对象，每次数据集发生变化时，都会使用数据集对象发送所有字段值（* 仅支持的字段）。

要使组件能够在数据集中检索记录，需要指定哪些字段为键字段，**示例：**若数据集中 ID 字段为键字段，则需编写如下代码

```
void OnAfterOpenDataSet(TDataSet *DataSet)
{
    DataSet->FieldByName("ID")->ProviderFlags =
        DataSet->FieldByName("ID")->ProviderFlags + [pfInKey];
}
```

组件

[TsgcWSPServer_Dataset](#)：服务器协议数据集 VCL 组件。

[TsgcWSPClient_Dataset](#)：客户端协议 Dataset VCL 组件。

[Javascript 组件](#)：客户端 Javascript 参考。

浏览器测试

如果您想用您喜欢的 Web 浏览器测试此协议，请输入此 URL（您需要定义自己的主机和端口）

<http://host:port/dataset.esegece.com.html>

TsgcWSPServer_Dataset

这是服务器协议数据集组件。

简介

这是服务器协议 Dataset 组件。您需要将此组件拖放到窗体上，使用 Server 属性选择一个 [TsgcWebSocketServer](#) 组件，并使用 Dataset 属性选择一个 Dataset 组件。

此组件继承自 [TsgcWSProtocol_Server_sgc](#) 的所有方法和属性。

参考

- 属性 — 组件上提供 13 个属性。
- 方法 — 9 个公共方法。
- 事件 — 可处理 22 个事件。

TsgcWSPClient_Dataset

这是客户端协议数据集组件。

简介

这是客户端协议数据集组件。您需要将此组件放置在表单上，并使用 Client 属性选择一个 [TsgcWebSocketClient](#) 组件，使用 Dataset 属性选择一个数据集组件。

本组件从 [TsgcWSProtocol_Client_sgc](#) 继承所有方法和属性。

参考

- Properties — 组件上共有 13 个可用属性。
- 方法 — 14 个公共方法。
- 事件 — 25 个可处理的事件。

TsgclWWSPClient_Dataset

这是 IntraWeb 客户端协议数据集组件。

简介

这是 IntraWeb 客户端协议数据集组件。您需要将此组件放置在表单上，使用 Client 属性选择一个 [TsgclWWebSocketClient](#) 组件，并使用 Dataset 属性选择一个数据集组件。

该组件继承自 [TsgclWWSPClient_sgc](#) 的所有方法和属性。

参考

- 属性 — 组件上有 3 个可用属性。
- 方法 — 6 个公开方法。
- 事件 — 9 个可处理的事件。

数据集协议 Javascript

Dataset 协议 Javascript：sgcWebSockets 使用 **sgcWebSocket.js** 和 **dataset.esegece.com.js** 文件。

您可以在此处找到可用的方法，需要根据实际情况替换 `{%host%}` 和 `{%port%}` 变量，例如：如果您将 sgcWebSocket 服务器配置为在 `www.example.com` 网站的 80 端口监听，则需要配置：

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/dataset.esegece.com.js"></script>
```

打开连接

```
<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>
```

发送消息

```
<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>
```

收到消息时显示提醒

```
<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>
```

显示已接收数据集的提示

```
<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>
```

订阅或取消订阅频道时显示警告

```
<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>
```

在 OnConnect、OnDisconnect 和 OnError 事件时显示警告

```
<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>
```

订阅所有数据集变更

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  socket.subscribe_all();
</script>
```

取消订阅所有 Dataset 变更

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  socket.unsubscribe_all();
</script>
```

处理数据集变更

```
<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>

```

关闭连接

```

<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
    socket.close();
</script>

```

获取连接状态

```

<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
    socket.state();
</script>

```

Protocol Dataset | 复制表

此模式尝试解决一个常见场景：一张表被所有已连接客户端复制，例如，如果您有一个带有股票报价表，您希望将广播股票变化给所有客户端，但您不希望客户端直接连接到您的数据库。因此，每次有变化时，记录信息将被广播给所有已连接客户端。每个客户端将读取记录并更新自己的表。

您可以在 Demos 文件夹中查看 SQLite/MultipleDatabase 演示。

配置数据集服务器

创建新的数据集协议服务器并使用以下属性进行配置：

- **ApplyUpdates**：设置为 **True**，每次发生变更时，都会广播给客户端
- **AutoSynchronize**：设为 **True**，每当新客户连接到服务器时，服务器将发送所有记录（元数据和数据），以便客户端获取服务器的最新信息。
- **UpdateMode**：设置为 **upWhereAll** 或 **upWhereChanged**。区别在于前者在更新时发送记录的所有字段，后者只发送已更改的字段。

```
TsgcWebSocketServer *oServer = new TsgcWebSocketServer();
TsgcWSPServer_Dataset *oProtocolDataset = new TsgcWSPServer_Dataset();
oProtocolDataset->Server = oServer;
oProtocolDataset->Dataset = <...your dataset...>;
oProtocolDataset->ApplyUpdates = true;
oProtocolDataset->AutoSynchronize = true;
oProtocolDataset->NotifyUpdates = true;
oProtocolDataset->UpdateMode = upWhereAll;
oServer->Port = 80;
oServer->Active = true;
```

配置 Dataset 客户端

创建一个新的数据集协议客户端并使用以下属性进行配置

- **ApplyUpdates**：设置为 **True** 时，每次发生更改都会将其发送到服务器。
- **AutoSubscribe**：设为 **True**，每当新客户连接到服务器时，客户端将自动订阅更新、删除和新记录。
- **UpdateMode**：设置为 **upWhereAll** 或 **upWhereChanged**。区别在于前者在更新时发送记录的所有字段，后者只发送已更改的字段。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSPClient_Dataset *oProtocolDataset = new TsgcWSPClient_Dataset();
oProtocolDataset->Client = oClient;
oProtocolDataset->Dataset = <...your dataset...>;
oProtocolDataset->ApplyUpdates = true;
oProtocolDataset->AutoSubscribe = true;
oProtocolDataset->NotifyUpdates = true;
oProtocolDataset->UpdateMode = upWhereAll;
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClient->Active = true;
```

协议数据集 | 通知更新

此模式尝试解决服务器和客户端共享单个数据库（服务器和客户端连接到同一物理数据库）且客户端希望在其他客户端对数据集进行任何更改时收到通知的场景。

您可以在 Demos 文件夹中查看 SQLite/SingleDatabase 演示。

配置数据集服务器

创建新的数据集协议服务器并使用以下属性进行配置：

- **ApplyUpdates**：设置为 **True**，每次发生变更时，都会广播给客户端
- **AutoSynchronize**：设为 **False**，此处无需设为 **True**，因为客户端连接的是与服务器相同的数据库。
- **UpdateMode**：设置为 **upRefreshAll**。

```
TsgcWebSocketServer *oServer = new TsgcWebSocketServer();
TsgcWSPServer_Dataset *oProtocolDataset = new TsgcWSPServer_Dataset();
oProtocolDataset->Server = oServer;
oProtocolDataset->Dataset = <...your dataset...>;
oProtocolDataset->ApplyUpdates = true;
oProtocolDataset->AutoSynchronize = false;
oProtocolDataset->NotifyUpdates = true;
oProtocolDataset->UpdateMode = upRefreshAll;
oServer->Port = 80;
oServer->Active = true;
```

配置 Dataset 客户端

创建一个新的数据集协议客户端并使用以下属性进行配置

- **ApplyUpdates**：设置为 **True** 时，每次发生更改都会将其发送到服务器。
- **AutoSubscribe**：设为 **True**，每当新客户端连接到服务器时，客户端将自动订阅更新、删除和新记录。
- **UpdateMode**：设置为 **upRefreshAll**。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSPClient_Dataset *oProtocolDataset = new TsgcWSPClient_Dataset();
oProtocolDataset->Client = oClient;
oProtocolDataset->Dataset = <...your dataset...>;
oProtocolDataset->ApplyUpdates = true;
oProtocolDataset->AutoSubscribe = true;
oProtocolDataset->NotifyUpdates = true;
oProtocolDataset->UpdateMode = upRefreshAll;
oClient->Host = "127.0.0.1";
oClient->Port = 80;
oClient->Active = true;
```

Protocol Files

此协议允许使用二进制 WebSocket 传输发送文件。可以以低内存占用处理大文件。

功能特性

- 发布/订阅消息模式，用于实现一对多消息分发和应用程序解耦。
- 确认已发送的消息。
- 实现 **QoS**（服务质量）用于文件传递。
- 可选择为接收的文件请求 **授权** 验证。
- 内存占用低。

组件

[TsgcWSPServer_Files](#)：服务器协议文件 VCL 组件。

[TsgcWSPClient_Files](#)：客户端协议文件 VCL 组件。

类

[TsgcWSMessageFile](#)：封装文件数据包信息的对象。

最常见的用途

- 发送文件
 - [如何向服务器发送文件](#)
 - [如何向客户端发送文件](#)
- 大文件
 - [如何发送大文件](#)

TsgcWSPServer_Files

这是服务器文件协议组件。

简介

这是服务器文件协议组件。您需要将此组件拖放到窗体中，并通过 Server 属性选择一个 [TsgcWebSocketServer](#) 组件。

参考

- 属性 — 组件上提供 5 个属性。
- 方法 — 3 个公共方法。
- 事件 — 20 个可处理的事件。

TsgcWSPClient_Files

这是客户端文件协议组件。

简介

这是客户端文件协议组件。您需要将此组件拖放到窗体中，并通过 Client 属性选择一个 [TsgcWebSocketClient](#) 组件。

参考

- 属性 — 组件上提供 5 个属性。
- 方法——4 个公共方法。
- 事件 — 19 个可处理的事件。

TsgcWSMessageFile

每次触发文件协议事件时，此对象将作为参数传递。

属性

- BufferSize：数据包的默认大小。
- Channel：如果指定，此文件将只发送给订阅了特定频道的客户端。
- 方法：内部方法。
- FileId：文件标识符，对所有已接收/已发送的文件唯一。
- Data：用户自定义数据。用户可在此设置任意文本。
- FileName：文件名称。
- FilePosition：文件位置（字节）。
- FileSize：文件总大小（字节）。
- Id：数据包的标识符，每个数据包唯一。
- QoS：消息的服务质量。
- Streaming: 供内部使用。
- Text：供内部使用。

协议文件 | 如何向服务器发送文件

要向服务器发送文件，只需调用 Files Protocol 的 **SendFile** 方法，并将完整的 **FileName** 作为参数传入即可。服务器收到的文件默认保存在服务器可执行文件所在的目录，或保存在 Files.SaveDirectory 属性中设置的路径下。

```
// ... Create Server
TsgcWebsocketServer oServer = new TsgcWebsocketServer();
TsgcWSPServer_Files oServer_Files = new TsgcWSPServer_Files();
oServer_Files->Server = oServer;
oServer->Host = "127.0.0.1";
oServer->Port = 8080;

// ... Create Client
TsgcWebsocketClient oClient = new TsgcWebsocketClient();
oClient->URL = "ws://127.0.0.1:8080";

// ... Create Protocol
TsgcWSPClient_Files oClient_Files = new TsgcWSPClient_Files();
oClient_Files->Client = oClient;

// ... Start Server
oServer->Active = true;

// ... Connect client and Send File
if oClient->Connect() then
    oClient_Files->SendFile("c:\Documents\yourfile.txt");
```

协议文件 | 如何向客户端发送文件

要向客户端发送文件，只需调用 Files 协议的 **SendFile** 方法，并传入连接的 **Guid** 和完整的 **FileName** 作为参数。客户端连接的 Guid 可在服务器协议 Files 的 OnConnect 事件中获取。客户端收到的文件默认将保存在客户端可执行文件所在的同一目录中，或保存在 Files.SaveDirectory 属性中设置的路径下。

```
// ... capture the guid of the client connection to send later the file
void OnConnectEvent(TsgcWSConnection *Connection)
{
    FGuid = Connection->Guid;
}

// ... Create Server
TsgcWebSocketServer oServer = new TsgcWebSocketServer();
TsgcWSPServer_Files oServer_Files = new TsgcWSPServer_Files();
oServer_Files->Server = oServer;
oServer_Files->OnConnect = OnConnectEvent;
oServer->Host = "127.0.0.1";
oServer->Port = 8080;

// ... Create Client
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient->URL = "ws://127.0.0.1:8080";

// ... Create Protocol
TsgcWSPClient_Files oClient_Files = new TsgcWSPClient_Files();
oClient_Files->Client = oClient;

// ... Start Server
oServer->Active = true;
oClient->Connect();

// ... Send File to the client connected
oServer_Files->SendFile(FGuid, "c:\\Documents\\yourfile.txt");
```

协议文件 | 如何发送大文件

当您想向服务器或客户端发送大文件（例如几个 GB 的文件）时，可能会在尝试加载完整文件时遇到内存问题。Protocol Files 允许您将文件分割成较小的数据包，对端收到后重新组装为单个文件。只需使用 **SendFile** 方法的 **Size** 参数设置每个数据包的大小（字节）。

```
// ... Create Server
TsgcWebSocketServer oServer = new TsgcWebSocketServer();
TsgcWSPServer_Files oServer_Files = new TsgcWSPServer_Files();
oServer_Files->Server = oServer;
oServer->Host = "127.0.0.1";
oServer->Port = 8080;

// ... Create Client
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient->URL = "ws://127.0.0.1:8080";

// ... Create Protocol
TsgcWSPClient_Files oClient_Files = new TsgcWSPClient_Files();
oClient_Files->Client = oClient;

// ... Start Server
oServer->Active = true;

// ... Connect client and Send File in packets of 100000 bytes
if oClient->Connect() then
    oClient_Files->SendFile("c:\Documents\yourfile.txt", 100000, qosLevel0, "");
```

Protocol Presence

Presence 协议允许您了解哪些用户订阅了某个频道，从而更方便地构建聊天应用并知晓谁在线，例如：游戏用户、聊天室、查看同一文档的用户.....

功能特性

- 默认情况下，用户**通过名称标识**，但可以通过传入更多数据进行**自定义**：邮箱、公司、Twitter 等。
- 事件 用于**授权**是否可以创建**频道**，是否允许**成员**加入.....
- 每当有新**成员加入**频道时，所有成员都会收到**通知**。
- 向**所有频道订阅者**发布消息。
- **内存占用低**。

组件

[TsgcWSPServer_Presence](#)：服务器协议 Presence VCL 组件。

[TsgcWSPClient_Presence](#)：客户端协议 Presence VCL 组件。

类

[TsgcWSPresenceMessage](#)：封装 presence 数据包信息的对象。

TsgcWSPServer_Presence

这是服务器 Presence 协议组件。

简介

这是服务器 Presence 协议组件。您需要将此组件拖放到窗体上，并使用 Server 属性选择一个 [TsgcWebSocketServer](#) 组件。

参考

- 属性 — 组件上可用的 6 个属性。
- 方法 — 1 个公共方法。
- 事件 — 17 个可处理的事件。

TsgcWSPresenceMessage

此对象封装服务器和客户端 Presence 协议交换的所有内部消息。

TsgcWSPresenceMember

ID：内部标识符

Name：成员名称，由客户端提供。

Info：成员附加信息，由客户端提供。

Data：可用于服务器用途的 TObject。

TsgcWSPresenceMemberList

Member[i]：按索引访问列表的成员

Count：列表的成员数量。

TsgcWSPresenceChannel

Name：频道名称，由客户端提供。

TsgcWSPresenceMsg

Text：文本消息，由客户端调用 Publish 方法时提供

TsgcWSPresenceError

Code：标识错误的整数值

Text：错误描述。

TsgcWSPClient_Presence

这是客户端在线状态协议组件。

简介

这是客户端 Presence 协议组件。您需要将此组件拖放到表单上，并使用 Client 属性选择一个 [TsgcWebSocketClient](#) 组件。

参考

- 属性 — 组件上可用的 7 个属性。
- 方法 — 5 个公共方法。
- 事件 — 16 个可处理的事件。

Protocol Presence Javascript

Presence 协议 JavaScript——sgcWebSockets 使用 **sgcWebSocket.js** 和 **presence.esegece.com.js** 文件。

在此可以找到可用的方法，您需要将 `{%host%}` 和 `{%port%}` 变量替换为实际值，例如：如果您已将 sgcWebSocket 服务器配置为在 `www.example.com` 网站的端口 80 上监听，则需要配置：

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/presence.esegece.com.js"></script>
```

打开连接

```
<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>
```

连接后的新成员

```
<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>
```

订阅主题 1 频道

```
<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>
```

取消订阅主题 1 通道

```
<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>
```

向主题 1 频道发布消息

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcsws_presence('ws://{%host%}:{%port%}');
  socket.publish('Hello sgcWebSockets!', 'Topic 1');
</script>
```

接收消息

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcsws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcpublishmsg', function(event)
  {
    console.log('#publishmsg: ' + event.message.text);
  });
</script>
```

获取所有已连接成员

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcsws_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>
```

当成员订阅/取消订阅时显示提醒

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcsws_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>
```

在 OnConnect、OnDisconnect 和 OnError 事件时显示警告

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.egegece.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>
```

关闭连接

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.egegece.com.js"></script>
<script>
  socket.close();
</script>
```

E2EE 协议

端对端加密（E2EE）意味着消息在发送方设备上加密，只能在接收方设备上解密。服务器路由数据包，但无法读取明文内容。

本主题解释以下内容的技术流程：

- **直接消息（1:1）** 在两个对端之间传递。
- **群组消息** 使用基于发送方密钥的模型，并在成员变更时轮换密钥。

密码学基础构建块

- **身份 / 密钥协商**：椭圆曲线 Diffie-Hellman（ECDH）P-256。
- **对称加密**：AES-256-GCM。
- **密钥派生**：HKDF-SHA-256，用于从共享密钥派生加密密钥。
- **消息认证**：AEAD 认证标签（由 AES-GCM 提供）。
- **重放保护**：每条消息使用 nonce/IV 并结合发送方序列计数器进行保护。

直接消息（1:1）技术流程

1. **公钥发现**
每个对等方发布一个公共身份密钥。服务器可以分发公钥，但私钥永远不会离开客户端设备。
2. **共享密钥建立**
发送方和接收方执行 ECDH（私钥 + 对等方公钥），获得相同的共享密钥。
3. **会话密钥派生**
HKDF-SHA-256 从 ECDH 输出派生一个或多个对称密钥（加密密钥、可选的头部密钥）。
4. **消息加密**
明文使用 AES-256-GCM 及唯一随机数/IV 加密。输出包含密文与认证标签。
5. **传输**
服务器在没有明文访问权限的情况下转发加密有效负载和元数据（例如：发送者 id、密钥 id、计数器、时间戳）。
6. **接收方解密**
接收方派生相同的会话密钥，验证认证标签并解密。任何篡改均会导致认证失败。

群组消息技术流程（发送方密钥）

对于群组，为每个成员单独加密每条消息的成本很高。一种常见的优化是**发送者密钥**设计：

- 每个发送方为每个群组维护一个**发送方密钥状态**。
- 该状态包含一个对称的**链密钥**和消息计数器。
- 对于每条出站群组消息，发送方从链密钥派生一次性消息密钥并推进链（哈希棘轮）。
- 消息密钥使用 AES-256-GCM 加密负载。

发件方密钥的分发方式

1. 当发送方加入群组时，会创建一个新的发送方密钥状态。
2. 发送方密钥状态通过现有的 1:1 加密会话（成对 E2EE）共享给每个当前成员。
3. 分发完成后，普通群组载荷使用发件人密钥快速路径（每条消息只需加密一次）。

成员变更与发送者密钥轮换

为保持前向和后向保密性，组发送者密钥必须在成员事件时轮换：

- **用户加入群组**：轮换发送方密钥，使新成员无法解密旧的历史记录，除非明确共享。
- **用户离开或被移除**：立即轮换发送方密钥，使被移除的成员无法解密新流量。
- **定期轮换**：可选的基于时间或消息计数的轮换，可降低密钥泄露的影响。

典型的轮换序列：

1. 创建新的发送方密钥状态（新密钥 ID，新链密钥）。
2. 通过成对的 E2EE 通道仅将其分发给当前已授权的成员。
3. 开始使用新密钥 id 加密新的组消息。
4. 仅在短暂过渡窗口内接受旧密钥 ID，然后将其停用。

安全属性

- **保密性**：只有具有有效密钥的客户端才能解密。
- **完整性与真实性**：AEAD 验证可检测篡改和伪造的密文。
- **服务器盲性**：中继服务器不持有消息内容的明文密钥。
- **成员离开后的保护**：密钥轮换可阻止前成员访问未来的群组消息。

组件

[TsgcWSPServer_E2EE](#)：服务器端 E2EE 协议组件。它在客户端之间转发加密消息，无法获知消息内容。

[TsgcWSPClient_E2EE](#)：客户端协议 E2EE 组件。负责在对等设备上管理密钥交换、加密和解密。

TsgcWSPServer_E2EE

E2EE 服务器协议组件在客户端之间路由加密的直接消息和群组消息，同时保持负载不透明。

简介

E2EE 服务器协议组件在客户端之间路由加密的直接消息和群组消息，同时保持负载不透明。将其放置在表单上，将 [TsgcWebSocketHTTPServer](#)（或其他服务器组件）分配给 **Server** 属性，并配置 E2EE 选项。实现位于单元 `sgcWebSocket_Protocol_E2EE_Server` 中。

配置

要使用此组件：

- 将 **Server** 设置为已配置的 **TsgcWebSocketHTTPServer** 实例。
- 在 **E2EE_Options.Ack** 中配置确认处理。
- 使用 **E2EE_Options.PublicKeys.ReceiveNewPublicKey** 选择处理新公钥的方式。
- 将 **Server.Active := True** 设置为开始监听。

参考

- 属性——组件上提供 5 个属性。
- 方法——0 个公开方法。
- 事件 — 8 个可处理的事件。

TsgcWSPClient_E2EE

E2EE 客户端协议组件在 WebSocket 连接上添加端到端加密功能，包括加密的直接消息和加密的群组消息。

简介

E2EE 客户端协议组件通过 WebSocket 连接添加端到端加密，包括加密的直接消息和加密的群组消息。将组件拖放到窗体上，将 [TsgcWebSocketClient](#) 分配给 **Client** 属性，并在连接之前配置 E2EE 选项。实现位于单元 `sgcWebSocket_Protocol_E2EE_Client` 中。

配置

要使用此组件：

- 将 **Client** 设置为已配置的 **TsgcWebSocketClient** 实例。
- 使用 **E2EE_Options.UserId** 分配唯一的用户标识符。
- 当您希望进行投递状态回调时，通过 **E2EE_Options.Ack.RcvDirectMessage** 和 **E2EE_Options.Ack.RcvGroupMessage** 启用消息确认。
- 将 **Client.Active := True** 设置为打开 WebSocket 连接。

参考

- 属性 — 组件上提供 5 个属性。
- 方法 — 9 个公共方法。
- 事件 — 19 个可处理的事件。

WebSocket APIs

基于 WebSocket 的实现有多种：金融、消息发布、队列等。sgcWebSockets 实现了基于 WebSocket 协议的最重要 API。使用 API 时，只需将 API 组件附加到客户端，所有消息将由 API 组件处理（每个客户端只能附加一个 API 组件）。

客户端 API

API	描述
Binance	是一家国际多语言加密货币交易所。
Binance Futures	允许您连接到 Binance Futures WebSocket / REST 市场数据流。
Coinbase	Coinbase 是一家美国加密货币交易所。可使用 USD、EUR 和 GBP 交易比特币 WebSocket API 和 REST API。
SignalR	是一个面向 ASP.NET 开发人员的库，使用 WebSocket 作为传输层，可简化实时 Web 应用。
SignalRCore	ASP.NET Core SignalR 是一个开源库，简化了向应用添加实时 Web 功能的过程。
SocketIO	是一个用于实时 Web 应用程序的 JavaScript 库。它实现了 Web 客户端和服务端之间的通信。
Kraken	是一家总部位于美国的加密货币交易所。
Kraken Futures	允许您连接到 Kraken Futures WebSocket / REST 市场数据。
Pusher	Pusher 是一个简单可靠的平台，提供灵活的发布/订阅消息传递、实时用户列表、认证和身份验证。
Bitfinex	Bitfinex 是全球最大、最先进的加密货币交易平台之一。用户可以交易比特币、以太坊、Lota、NEO、莱特币、以太坊经典.....
Bitstamp	Bitstamp 是全球历史最悠久的加密货币交易所之一，自 2011 年起支持区块链生态系统。
Huobi	是一家国际多语言加密货币交易所。
Cex	是一个加密货币交易所，也是前比特币云挖矿提供商。
Cex Plus	CEX.IO Exchange Plus 是终极加密货币交易平台，具备深度流动性和高级交易工具。
Bitmex	是一个加密货币交易所和衍生品交易平台。
3Commas	这是一个加密交易机器人。
Kucoin	是一个加密货币交易所，允许您购买、出售和存储 BTC、ETH、KCS、SHIB、DOGE 等。
Kucoin Futures	允许您连接到 Kucoin 期货服务器（WebSocket 和 REST）
OKX	前身为 OKEx，是全球最大的加密现货和衍生品交易所之一。
XTB	外汇和差价合约交易，提供对超过 2000 个金融市场的访问。
Discord	是目前最受欢迎的在线游戏和流媒体通讯工具之一。
Bybit	加密货币交易所和交易平台
OpenAI API	OpenAI Realtime API 实现低延迟、多模态交互，包括语音到语音对话体验和实时转录。
MEXC	中心化加密货币交易所和交易平台，本组件实现了 SPOT API。
MEXC Futures	中心化加密货币交易所和交易平台，此组件实现 Futures API。
Bitget	支持现货和期货市场的加密货币交易所和交易平台。
Gate.io	支持现货和期货交易的加密货币交易所，提供 WebSocket 和 REST API。
Deribit	提供比特币和以太坊期货及期权交易的加密货币衍生品交易所。
Crypto.com	支持通过 WebSocket 和 REST API 进行市场和用户频道的加密货币交易所。
HTX	国际加密货币交易所（原火币），提供用于市场数据和交易的 REST API。

WebSocket API 可在运行时注册，只需调用 **RegisterAPI** 方法并将 API 组件作为参数传入即可。

其他客户端 API

API	描述
Telegram	是一种基于云的即时通讯和网络语音服务。用户可以发送消息，交换照片、视频、贴图。
Whatsapp	是一款国际通用的美国免费跨平台集中式即时通讯和网络电话服务。
RCON	是一种基于 TCP/IP 的通信协议，允许向服务器发出控制台命令。
CryptoHopper	这是一个加密交易机器人和投资组合管理器。
CryptoRobotics	这是一个加密交易机器人。

服务器 API

API	描述
RTCMultiConnection	RTCMultiConnection 是一个用于点对点应用（屏幕共享、音视频会议、文件传输）的 JavaScript 库。
WebPush	Web Push 协议允许 Web 应用程序即使在应用程序未打开或未激活时也能向用户推送通知。
WebAuthn	FIDO2/WebAuthn 服务器 API，用于通过生物识别、安全密钥和通行密钥进行身份验证。

API Binance

Binance

Binance 是一家国际多语言加密货币交易所。它提供了一些 API 来访问 Binance 数据。支持以下 API：

1. **WebSocket 流**：允许您订阅某些方法并实时获取数据。服务器将事件推送给订阅者。使用 WebSocket 作为协议。
2. **UserData 流**：已订阅的客户端获取账户详情。需要 API 密钥进行身份验证，并使用 WebSocket 作为协议。
3. **REST API**：需要 API 密钥和密钥进行身份验证，使用 HTTPS 作为协议。
 1. [市场数据](#)
 2. [账户和交易数据](#)
 3. [钱包](#)
4. **期货**：WebSocket 期货市场数据流通过 [Binance Futures 客户端 API](#) 提供支持。

该客户端也支持 [Binance.us](#)，以下 API 受到支持：

1. **WebSocket 流**：允许您订阅某些方法并实时获取数据。服务器将事件推送给订阅者。使用 WebSocket 作为协议。
2. **UserData 流**：已订阅的客户端获取账户详情。需要 API 密钥进行身份验证，并使用 WebSocket 作为协议。
3. **REST API**：客户端可向服务器请求市场和账户数据。需要 API Key 和 Secret 进行身份验证，并使用 HTTPS 协议。

属性

Binance API 有两种方法：公开和私有。公开方法无需身份验证即可访问，例如获取行情价格。部分与用户数据相关的方法是私有的，需要使用 Binance API 密钥。

- **ApiKey**：您可以在 Binance 账户中申请新的 API 密钥，只需将其值复制到此属性即可。
- **ApiSecret**：API 密钥仅在 REST_API 中需要，WebSocket API 中只有部分方法需要 ApiKey。
- **TestNet**：如果启用，将连接到 Binance 演示账户（默认为 false）。
 - **HTTPLogOptions**：在文本文件中存储 HTTP 请求日志
 - **Enabled**：如果启用，将存储 WebSocket API 的所有 HTTP 请求。
 - **FileName**：存储日志的文件完整路径
 - **REST**：在文本文件中存储 REST API 请求日志
 - **Enabled**：如果启用，将存储 REST API 的所有 HTTP 请求。
 - **FileName**：存储日志的文件完整路径。
- **UserStream**：若启用，客户端将接收账户、订单或余额更新通知（默认为 true）。
- **BinanceUS**：如果启用，将连接到 [Binance.us](#) 服务器（而非默认的 [Binance.com](#) 服务器）。
- **ListenKeyOnDisconnect**：此属性指定当客户端从 Binance 服务器断开连接且存在活动 ListenKey 时的处理方式。
 - **blkodDeleteListenKey**：通过向 Binance 服务器发送 HTTP 请求来删除活动 ListenKey（这是默认行为）。
 - **blkodClearListenKey**：不从 Binance 服务器删除 ListenKey，仅清除该字段的值。
 - **blkodDoNothing**：不执行任何操作，下次连接到 Binance 时将尝试使用相同的 ListenKey。
- **UseCombinedStreams**：若启用，将按如下方式合并流：{"stream":"<streamName>","data":<rawPayload>}（默认禁用）

最常见的用途

- **WebSockets API**
 - [如何连接到 WebSocket API](#)
 - [如何订阅 WebSocket 频道](#)
- **REST API**
 - [如何获取市场数据](#)
 - [如何使用私有 REST API](#)

- 如何进行现货交易
- 私有请求时间
- 提现

WebSocket 流 API

基础端点为 `wss://stream.binance.com:9443`，客户端可在成功连接后订阅/取消订阅事件。
支持以下订阅/取消订阅方法。

方法	参数	描述
AggregateTrades	交易对	推送为单个 taker 订单聚合的交易信息
交易	交易对	推送原始交易信息；每笔交易有唯一的买方和卖方
KLine	Symbol, Interval	每秒、每分钟、每小时...推送当前 K 线/烛线更新。
MiniTicker	交易对	24 小时滚动窗口迷你行情统计。这些数据不是 UTC 日的统计数据，而是前 24 小时的 24 小时滚动窗口数据。
AllMiniTickers		所有价格变动交易对的 24 小时滚动窗口迷你 ticker 统计数据（以数组形式）。这些不是 UTC 日内的统计数据，而是过去 24 小时的 24 小时滚动窗口数据。请注意，数组中只包含价格发生变动的 ticker。
Ticker	交易对	单个交易对的 24 小时滚动窗口行情统计。这不是 UTC 天的统计数据，而是前 24 小时的滚动窗口。
AllMarketTickers		所有发生变化的交易对 24 小时滚动窗口行情统计数组。这不是 UTC 天的统计数据，而是前 24 小时的滚动窗口。请注意，数组中只会包含发生变化的行情。
BookTicker	交易对	实时推送指定交易对最优买卖价格或数量的任何更新。
AllBookTickers		实时推送所有交易对的最优买卖价或数量的任何更新。
PartialBookDepth	交易对，深度	前 <levels> 档买卖盘，每秒推送一次。有效的 <levels> 值为 5、10 或 20。
DiffDepth	交易对	用于在本地管理订单簿的订单簿价格和数量深度更新。

成功订阅/取消订阅后，客户端会收到一条相关消息，其中 id 是 Subscribed / Unsubscribed 方法的返回结果。

```
{
  "result": null,
  "id": 1
}
```

User Data Stream API

需要从您的 Binance 账户获取有效的 ApiKey，并将 ApiKey 设置到组件的 Binance.ApiKey 属性中。

每次发生变化时，以下数据将被推送给客户端。如果设置了有效的 ApiKey，无需订阅任何方法，系统将自动完成订阅。

方法	描述
账户更新	账户状态通过 outboundAccountInfo 事件进行更新。
余额更新	余额更新发生在以下情况下： • 账户存款或取款 • 账户间资金转账（例如现货账户转至保证金账户）

订单更新

订单通过 executionReport 事件更新。

REST API

基础端点为：<https://api.binance.com>。所有端点返回 JSON 对象或数组。数据按升序返回，最旧的在前，最新的在后。

使用 `REST_API.BinanceOptions` 属性访问 REST API 选项。

公共 API 端点

这些端点无需任何授权即可访问。

通用端点

方法	参数	描述
Ping		测试与 Rest API 的连通性。
GetServerTime		测试与 Rest API 的连接性并获取当前服务器时间。
GetExchangeInformation		当前交易所交易规则和交易对信息

市场数据端点

方法	参数	描述
GetOrderBook	交易对	获取订单簿。
GetTrades	交易对	获取最新成交
GetHistoricalTrades	交易对	获取历史交易记录。
GetAggregateTrades	交易对	获取压缩的聚合交易数据。在同一时间以相同价格成交的同一订单的交易，其数量将被聚合。
GetKLines	Symbol, Interval	某交易对的 K 线/烛线柱。K 线以其开盘时间唯一标识。
GetAveragePrice	交易对	某个交易对的当前均价。
Get24hrTicker	交易对	24 小时滚动窗口价格变动统计。不带 symbol 参数访问时请谨慎。
GetPriceTicker	交易对	某个交易对的最新价格。
GetPriceTickers	Symbols	一个交易对或多个交易对的最新价格。示例： ["BTCUSDT", "BNBUSDT"]
GetBookTicker	交易对	某个或多个交易对在订单簿中的最优价格/数量。
GetUIKLines	Symbol, Interval	某一交易对的 K 线/烛台图数据。响应格式与 GetKLines 类似，针对烛台图展示进行了优化。
GetRollingWindowTicker	Symbol, Symbols, WindowSize	滚动窗口价格变动统计。注意：如果未指定，WindowSize 默认为 1d。
GetTradingDayTicker	Symbol, Symbols, Type	某个交易日的价格变动统计数据。

私有 API 端点

需要 APIKey 和 APISecret 才能获得服务器授权。

账户数据端点

方法	参数	描述
NewOrder	Symbol, Side, Type	提交新订单。
PlaceMarketOrder	Side, Symbol, Quantity	下新的市价单
PlaceMarketQuoteOrder	Side, Symbol, QuoteOrderQty	下新市价报价单
PlaceLimitOrder	Side, Symbol, Quantity, LimitPrice	下限价单
PlaceStopOrder	方向、交易对、数量、止损价格、限价价格	下新止损单
PlaceStopTrailingOrder	方向, 交易对, 数量, TrailingDelta, LimitPrice	下新的追踪止损订单
PlaceTakeProfitOrder	方向、交易对、数量、止损价格、限价价格	下新的止盈订单
PlaceTakeProfitTrailingOrder	方向, 交易对, 数量, TrailingDelta, LimitPrice	下达新的止盈跟踪止损订单
PlaceLimitMakerOrder	Side, Symbol, Quantity	下新的限价市价单
TestNewOrder	Symbol, Side, Type	测试新订单创建和签名/recvWindow 时长。创建并验证新订单, 但不将其发送到撮合引擎。
QueryOrder	交易对	查询订单状态。
CancelOrder	交易对	取消活跃订单。取消活跃订单。必须发送 OrderId 或 OrigClientOrderId 之一。
CancelAllOpenOrders	Symbol (可选)	
GetOpenOrders		获取某个交易对的所有未结订单。访问时不指定交易对请谨慎。
GetAllOrders	交易对	获取所有账户订单; 包括活跃、已取消或已成交的订单。
NewOCO	Symbol, Side, Quantity, Price, StopPrice	发送新的 OCO 订单
CancelOCO	交易对	取消整个订单列表
QueryOCO	交易对	根据提供的可选参数检索特定 OCO 订单
GetAllOCO		根据提供的可选参数检索所有 OCO 订单
GetOpenOCO		获取所有未完成的 OCO 订单。
GetAccountInformation		获取当前账户信息。
GetAccountTradeList	交易对	获取特定账户和交易对的成交记录。
CancelReplaceOrder	Symbol, Side, Type, CancelReplaceMode	取消现有订单并在同一交易对上下新订单。
NewOrderListOCO	Symbol, Side, Quantity, AboveType, BelowType	下新的 OCO 订单列表。
NewOrderListOTO	Symbol, WorkingType, WorkingSide, WorkingQuantity, WorkingPrice, PendingType, PendingSide, PendingQuantity	下一个新的 OTO (一触即发) 订单列表。
NewOrderListOTOCO	Symbol, WorkingType, WorkingSide, WorkingQuantity,	下达新的 OTOCO (一触发一取消另一个) 订单列表。

	WorkingPrice, PendingSide, PendingAboveType, PendingBelowType, PendingQuantity	
NewSOROrder	Symbol, Side, Type, Quantity	使用智能订单路由（SOR）下单。
TestSOROrder	Symbol, Side, Type, Quantity	使用智能订单路由（SOR）测试新订单。创建并验证新订单，但不将其发送至撮合引擎。
GetOrderRateLimitUsage		显示用户所有时间间隔的当前订单数使用情况。
GetPreventedMatches	交易对	显示因 STP（自成交防护）而过期的订单列表。
GetAllocations	交易对	检索 SOR 订单执行产生的分配。
GetAccountCommission	交易对	获取当前账户手续费率。

转换端点

方法	参数	描述
GetAllConvertPairs	FromAsset, ToAsset	查询所有可转换代币对各代币的上限/下限
GetConvertAssetInfo		查询支持资产的精度信息
SendConvertQuoteRequest	FromAsset, ToAsset	为请求的令牌对请求报价
AcceptConvertQuote	QuoteId	通过报价 ID 接受所提供的报价。
GetConvertOrderStatus	OrderId 或 QuoteId	按订单 ID 查询订单状态。
PlaceConvertLimitOrder	BaseAsset, QuoteAsset, Side, LimitPrice	允许用户下限价单。 baseAsset 或 quoteAsset 可通过 exchangeInfo 端点确定。 限价由 baseAsset 到 quoteAsset 定义。 使用 baseAmount 或 quoteAmount 其中之一。
CancelConvertLimitOrder	OrderId	允许用户取消限价订单
GetConvertLimitOpenOrders		允许用户查询所有现有限价单
GetConvertTradeHistory	StartTime, EndTime	startTime 与 endTime 之间的最大间隔为 30 天。

钱包端点

(*钱包端点仅适用于生产服务器，不适用于演示服务器)

方法	描述
GetWalletSystemStatus	获取系统状态。
GetWalletAllCoinsInformation	获取用户可用于存入和提取的代币信息。
GetWalletDailyAccountSnapshot	类型："SPOT"、"MARGIN"、"FUTURES" - 查询时间段必须少于 30 天 - 仅支持查询最近一个月内的数据 - 如果未发送 startTime 和 endTime，默认返回最近 7 天的记录
SetWalletDisableFastWithdrawSwitch	此请求将禁用您账户下的快速提现开关。 您需要为请求此端点的 API 密钥启用"交易"选项。
SetWalletEnableFastWithdrawSwitch	此请求将在您的账户下启用快速提现开关。 您需要为请求此端点的 API 密钥启用"交易"选项。 当快速提现开关开启后，向 Binance 账户划转资金将即时完成，无需链上交易，无交易 ID，也无提现手续费。
WalletWithdraw	提交提款请求。

GetWalletDepositHistory	获取存款历史。
GetWalletWithdrawHistory	获取提现历史记录。
GetWalletDepositAddress	通过网络获取充值地址。
GetWalletAccountStatus	获取账户状态详情。
GetWalletAccountAPITradingStatus	获取账户 API 交易状态详情。
GetWalletDustLog	仅返回最后 100 条记录 仅返回 2020/12/01 之后的记录
GetWalletAssetsConvertedBNB	
WalletDustTransfer	将零散资产转换为 BNB。 您需要为请求此端点的 API Key 开启现货与杠杆交易权限。
GetWalletAssetDividendRecord	查询资产分红记录。
GetWalletAssetDetail	获取 Binance 支持的资产详情。
GetWalletTradeFee	获取交易手续费
WalletUserUniversalTransfer	您需要为请求此端点的 API Key 启用"允许通用划转"选项。 MAIN_UMFUTURE 现货账户划转至 USDⓈ-M 合约账户 类型的枚举： - MAIN_CMFUTURE 现货账户划转至 COIN-M 合约账户 - MAIN_MARGIN 现货账户划转至保证金（全仓）账户 - UMFUTURE_MAIN USDⓈ-M 合约账户划转至现货账户 - UMFUTURE_MARGIN USDⓈ-M 合约账户划转至保证金（全仓）账户 - CMFUTURE_MAIN COIN-M 合约账户划转至现货账户 - CMFUTURE_MARGIN COIN-M 合约账户划转至保证金（全仓）账户 - MARGIN_MAIN 保证金（全仓）账户划转至现货账户 - MARGIN_UMFUTURE 保证金（全仓）账户划转至 USDⓈ-M 合约 - MARGIN_CMFUTURE 保证金（全仓）账户划转至 COIN-M 合约 - ISOLATEDMARGIN_MARGIN 逐仓保证金账户划转至保证金（全仓）账户 - MARGIN_ISOLATEDMARGIN 保证金（全仓）账户划转至逐仓保证金账户 - ISOLATEDMARGIN_ISOLATEDMARGIN 逐仓保证金账户划转至逐仓保证金账户 - MAIN_FUNDING 现货账户划转至资金账户 - FUNDING_MAIN 资金账户划转至现货账户 - FUNDING_UMFUTURE 资金账户划转至 UMFUTURE 账户 - UMFUTURE_FUNDING UMFUTURE 账户划转至资金账户 - MARGIN_FUNDING MARGIN 账户划转至资金账户 - FUNDING_MARGIN 资金账户划转至保证金账户 - FUNDING_CMFUTURE 资金账户划转至 CMFUTURE 账户 - CMFUTURE_FUNDING CMFUTURE 账户划转至资金账户
GetWalletQueryUserUniversalTransferHistory	- 当 type 为 ISOLATEDMARGIN_MARGIN 和 ISOLATEDMARGIN_ISOLATEDMARGIN 时，必须发送 fromSymbol - 当 type 为 MARGIN_ISOLATEDMARGIN 和 ISOLATEDMARGIN_ISOLATEDMARGIN 时，必须发送 toSymbol - 仅支持查询最近 6 个月内的数据 - 如果未发送 startTime 和 endTime，默认返回最近 7 天的记录
GetWalletFundingWallet	目前支持查询以下业务资产：Binance Pay、Binance Card、Binance Gift Card、Stock Token
GetWalletUserAsset	获取用户资产，仅返回正值数据。
GetWalletApiKeyPermission	

事件

Binance 消息在 TsgcWebSocketClient 组件中接收，您可以使用以下事件：

OnConnect

成功连接到 Binance 服务器后。

OnDisconnect

与 Binance 服务器断开连接后

OnMessage

服务器发送给客户端的消息在此事件中处理。

OnError

如果协议出现任何错误，将调用此事件。

OnException

如果存在未处理的异常，将调用此事件。

此外，在 Binance API 组件中有一个名为 **OnBinanceHTTPException** 的特定事件，每次调用 HTTP 请求（REST API 或 WebSocket 用户数据流）出错时触发。

(*) 由于 Binance 服务器的变更，Rad Studio 10.1 之前的 Indy 版本将无法连接到测试服务器。此问题不影响 Enterprise 版，或已升级到最新 Indy 版本的情况。

Binance | 连接 WebSocket API

要连接到 Binance WebSocket API，只需创建一个新的 Binance API 客户端并附加到 TsgcWebSocketClient。请参见下方示例：

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(NULL);
oBinance->Client = oClient;
oClient->Active = true;
```

Binance | 订阅 WebSocket 频道

Binance 提供多种频道供您订阅，以获取市场数据、订单等的实时更新。以下是订阅 Ticker 的示例：

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(NULL);
oBinance->Client = oClient;
oBinance->SubscribeTicker("bnbbtc");

void OnMessage(TsgcWSConnection *Connection, const string aText)
{
    // here you will receive the ticker updates
}
```

Binance | 获取市场数据

Binance 通过 REST 端点提供公共市场数据，调用这些端点时，您将获得所请求市场数据的快照。市场数据端点不需要身份验证，所有用户均可免费访问。

示例：若要获取 BNBBTC 的行情快照，请进行以下调用：

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
ShowMessage(oBinance->REST_API->GetPriceTicker("BNBBTC"));
```

Binance | 私有 REST API

Binance REST API 提供公开和私有端点。私有端点要求对消息进行签名以提高交易安全性。首先，您必须登录您的 Binance 账户并创建新的 API，您将获得以下值：

- ApiKey
- ApiSecret

这些字段必须在 Binance API 客户端组件的 Binance 属性中进行配置。配置完成后，您可以开始向 Binance Pro REST API 发出私有请求

*私有请求要求您的本地计算机时间已同步，否则请求将被 Binance 服务器拒绝。请查看以下文章，[Binance 私有请求时间](#)。

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
oBinance->Binance->ApiKey = "<your api key>";
oBinance->Binance->ApiSecret = "<your api secret>";
ShowMessage(oBinance->REST_API->GetAccountInformation());
```

Binance | 现货交易

Binance 允许您使用其 REST API 进行现货交易。

配置

首先，您必须在 Binance 账户中创建一个 **API Key**，并添加现货交易权限。

完成此操作后，您可以开始现货交易。

首先，在 **Binance 客户端组件**中设置您的 **ApiKey** 和 **ApiSecret**，这将用于签署发送至 Binance 服务器的请求。

下达订单

要下新订单，只需调用 Binance 客户端组件的 **REST_API.NewOrder** 方法。

根据订单类型（市价、限价等），API 需要提供的字段数量多少不等。

必填字段

- **Symbol**：产品 id 符号，示例：BNBBTC
- **方向**：买入或卖出
- **type**：订单类型
 - LIMIT
 - MARKET
 - STOP_LOSS
 - STOP_LOSS_LIMIT
 - TAKE_PROFIT
 - TAKE_PROFIT_LIMIT
 - LIMIT_MAKER

基于类型的其他必填字段

- **LIMIT**：timeInForce、quantity、price
- **MARKET**：quantity 或 quoteOrderQty
- **STOP_LOSS / TAKE_PROFIT**：quantity, stopPrice
- **STOP_LOSS_LIMIT / TAKE_PROFIT_LIMIT**：timeInForce, quantity, price, stopPrice
- **LIMIT_MAKER**：数量、价格

下订单时，有 2 种可能性：

1. **成功**：函数 NewOrder 返回 Binance 服务器发送的消息。
2. **错误**：异常将在 OnBinanceHTTPException 事件中返回。

以市价下单购买 1 个 BNBBTC

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
oBinance->Binance->ApiKey = "<api key>";
oBinance->Binance->ApiSecret = "<api secret>";
ShowMessage(oBinance->REST_API->NewOrder("BNBBTC", "BUY", "MARKET", "", 1));
```

以 0.009260 的价格下单购买 1 个 BNBBTC

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
oBinance->Binance->ApiKey = "<api key>";
oBinance->Binance->ApiSecret = "<api secret>";
ShowMessage(oBinance->REST_API->NewOrder("BNBBTC", "BUY", "LIMIT", "GTC", 1, 0, 0.009260));
```

-

Binance | 私有请求时间

当您向 Binance 发出私有请求时，消息会被签名以提高请求的安全性。消息包含本地时间并发送在签名消息中，如果本地时间与 Binance 服务器的差异超过 5 秒，请求将被拒绝。因此，验证本地时间是否已同步非常重要，您可以使用操作系统的时间同步方法来实现。

逻辑如下：

```
if (timestamp < (serverTime + 1000) && (serverTime - timestamp) <= recvWindow) {  
    // process request  
} else {  
    // reject request  
}
```

建议使用较小的 `recvWindow`（5000 或更小）！最大值不能超过 60000 毫秒。

您可以通过调用方法 **GetServerTime** 查看 Binance 服务器时间，该方法将返回 Binance 服务器的时间。

RecvWindow 默认值为 **5000**，可通过属性 **REST_API.BinanceOptions.RecvWindow** 增大该值。

Binance | 提现

Binance 允许您使用 Wallet API 提交提现请求，只有以下参数为必填项：

- 币种
- 地址
- 金额

```
TsgcWSAPI_Binance *oBinance = new TsgcWSAPI_Binance(this);
oBinance->Binance->ApiKey = "<your api key>";
oBinance->Binance->ApiSecret = "<your api secret>";
ShowMessage(oBinance->REST_API->WalletWithdraw("BTC", "7213fea8e94b4a5593d507237e5a555b", 0.25));
```


API Binance Futures

Binance

Binance 是一家国际多语言加密货币交易所。它提供了一些 API 用于访问 Binance 数据。此组件允许您获取 Binance 期货 WebSocket 市场流。

<https://binance-docs.github.io/apidocs/futures/en>

<https://binance-docs.github.io/apidocs/delivery/en>

期货合约

Binance API 有两种类型的方法：公开和私有。公开方法无需身份验证即可访问，例如：获取行情价格。部分方法是私有的，与用户数据相关；这些方法需要使用 Binance API 密钥。

- **ApiKey**：您可以在 Binance 账户中申请新的 API 密钥，只需将其值复制到此属性即可。
- **ApiSecret**：API 密钥仅在 REST_API 中需要，WebSocket API 中只有部分方法需要 ApiKey。
- **TestNet**：如果启用，将连接到 Binance 演示账户（默认为 false）。
 - **HTTPLogOptions**：在文本文件中存储 HTTP 请求日志
 - **Enabled**：如果启用，将存储 WebSocket API 的所有 HTTP 请求。
 - **FileName**：存储日志的文件完整路径
 - **REST**：在文本文件中存储 REST API 请求日志
 - **Enabled**：如果启用，将存储 REST API 的所有 HTTP 请求。
 - **FileName**：存储日志的文件完整路径。
- **UserStream**：若启用，客户端将接收账户、订单或余额更新通知（默认为 true）。
- **ListenKeyOnDisconnect**：此属性指定当客户端从 Binance 服务器断开连接时，若存在活跃的 ListenKey 应执行的操作。
 - **blkodDeleteListenKey**：通过向 Binance 服务器发送 HTTP 请求删除活跃的 ListenKey（这是默认行为）。
 - **blkodClearListenKey**：不从 Binance 服务器删除 ListenKey，仅清除该字段的值。
 - **blkodDoNothing**：不执行任何操作，下次连接到 Binance 时将尝试使用相同的 ListenKey。
- **UseCombinedStreams**：若启用，将按如下方式合并流：{"stream":"<streamName>","data":<rawPayload>}（默认禁用）

客户端可以连接到 **USDT** 或 **COIN** Binance Futures，使用 **FuturesContracts** 属性设置您要交易的合约类型：

- **bfcUSDT**：连接到 USD-M 期货 API。
- **bfcCOIN**：连接到 COIN-M Futures API。

客户端可以连接到生产或演示 Binance 账户。如果启用了 **TestNet** 属性，将连接到演示账户，否则将连接到生产 Binance 服务器。

WebSocket 流 API

客户端可以在成功连接后订阅/取消订阅事件。
支持以下订阅/取消订阅方法。

方法	参数	描述
AggregateTrades	交易对	聚合交易流每 100 毫秒推送一次针对单个吃单订单的聚合交易信息。
MarkPrice	Symbol, UpdateSpeed	单个交易对的标记价格和资金费率，每 3 秒或每秒推送一次。
AllMarkPrice	UpdateSpeed	所有交易对的标记价格和资金费率每 3 秒或每秒推送一次。
KLine	Symbol, Interval	K 线/蜡烛图流每 250 毫秒推送一次当前 K 线/蜡烛图的更新（如果存在）。

MiniTicker	交易对	单个品种的 24 小时滚动窗口迷你行情统计。这不是 UTC 日的统计数据，而是从 requestTime 到 24 小时前的 24 小时滚动窗口。
AllMiniTicker		所有交易对的 24 小时滚动窗口迷你行情统计。这些不是 UTC 日统计，而是从请求时间到之前 24 小时的滚动窗口。请注意，数组中只会包含已发生变化的行情。
Ticker	交易对	单个交易对的 24 小时滚动窗口行情统计。这不是 UTC 日的统计数据，而是从 requestTime 到 requestTime 之前 24 小时的滚动窗口统计。
AllMarketTickers		所有交易对的 24 小时滚动窗口行情统计。这些不是 UTC 日统计数据，而是从请求时间到 24 小时前的滚动窗口。请注意，数组中只会包含已发生变化的行情。
BookTicker	交易对	实时推送指定交易对最优买卖价格或数量的任何更新。
AllBookTickers		实时推送所有交易对的最优买卖价或数量的任何更新。
LiquidationOrders	交易对	强制平仓订单流推送特定交易对的强制平仓订单信息
AllLiquidationOrders		所有强制平仓订单流推送市场中所有交易对的强制平仓订单信息。
PartialBookDepth	交易对，深度	最优买卖报价，有效值为 5、10 或 20。
DiffDepth	交易对	买卖盘每 250 毫秒、500 毫秒、100 毫秒推送一次，或实时推送（如有）

成功订阅/取消订阅后，客户端会收到相关消息，其中 id 是 Subscribed / Unsubscribed 方法的结果。

```
{
  "result": null,
  "id": 1
}
```

User Data Stream API

需要从您的 Binance 账户获取有效的 ApiKey，并将 ApiKey 设置到组件的 Binance.ApiKey 属性中。

每次发生变化时，以下数据将被推送给客户端。如果设置了有效的 ApiKey，无需订阅任何方法，系统将自动完成订阅。

方法	描述
追加保证金通知	当用户的持仓风险率过高时，此流将被推送。此消息仅作为风险提示信息，不建议作为投资策略依据。在市场波动剧烈的情况下，推送此流时，用户的持仓可能已同时被强制平仓。
余额与持仓更新	余额更新发生在以下情况下： - 当余额或持仓更新时，将推送此事件。 - 当“资金费用”变更至用户余额时。
订单更新	创建新订单时，订单状态变更将推送此类事件。

REST API

所有端点均返回 JSON 对象或数组。数据按升序返回，最旧的在前，最新的在后。

公共 API 端点

这些端点无需任何授权即可访问。

通用端点

方法	参数	描述
Ping		测试与 Rest API 的连通性。
GetServerTime		测试与 Rest API 的连接性并获取当前服务器时间。
GetExchangeInformation		当前交易所交易规则和交易对信息

市场数据端点

方法	参数	描述
GetOrderBook	交易对	获取订单簿。
GetTrades	交易对	获取最新成交
GetHistoricalTrades	交易对	获取历史交易记录。
GetAggregateTrades	交易对	获取压缩的聚合交易数据。在同一时间以相同价格成交的同一订单的交易，其数量将被聚合。
GetKLines	Symbol, Interval	某交易对的 K 线/蜡烛线柱。K 线以其开盘时间唯一标识。
Get24hrTicker	交易对	24 小时滚动窗口价格变动统计。不带 symbol 参数访问时请谨慎。
GetPriceTicker	交易对	某个或多个标的的最新价格。
GetBookTicker	交易对	某个或多个交易对在订单簿中的最优价格/数量。
GetMarkPrice	交易对	标记价格和资金费率
GetFundingRateHistory	交易对	
GetOpenInterest	交易对	获取特定品种当前的持仓量。
GetOpenInterestStatistics	Symbol, Period	
GetTopTraderAccountRatio	Symbol, Period	
GetTopTraderPositionRatio	Symbol, Period	
GetGlobalAccountRatio	Symbol, Period	
GetTakerVolume	Symbol, Period	
GetContinuousKLines	Pair, ContractType, Interval	特定合约类型的 K 线/蜡烛图数据。
GetIndexPriceKLines	Pair, Interval	交易对指数价格的 K 线/蜡烛图数据。
GetMarkPriceKLines	Symbol, Interval	某交易对标记价格的 K 线/蜡烛图数据。
GetPremiumIndexKLines	Symbol, Interval	某个交易对的溢价指数 K 线。
GetFundingInfo		获取所有品种的资金费率信息。
GetPriceTickerV2	交易对	某个交易对或多个交易对的最新价格（V2）。
GetIndexInfo	交易对	获取索引信息。
GetAssetIndex	交易对	获取多资产模式的资产指数。
GetConstituents	交易对	获取指数成分。
GetDeliveryPrice	货币对	获取交割价格。
GetBasis	Pair, ContractType, Period	获取基础数据。

私有 API 端点

需要 APIKey 和 APISecret 才能获得服务器授权。

账户与交易端点

方法	参数	描述
ChangePositionMode	DualPosition	更改用户在所有交易对上的持仓模式（对冲模式或单向模式）
GetCurrentPositionMode		获取用户在所有交易对上的持仓模式（对冲模式或单向模式）
NewOrder	Symbol、Side、PositionSide、Type	提交新订单。
PlaceMarketOrder	Side, Symbol, Quantity	
PlaceLimitOrder	Side, Symbol, Quantity, LimitPrice	
PlaceStopOrder	方向、交易对、数量、止损价格、限价价格	
PlaceTrailingStopOrder	Side, Symbol, Quantity, aActivationPrice, aCallbackRate	
QueryOrder	交易对	查询订单状态。
CancelOrder	交易对	取消一个活跃订单。必须发送 OrderId 或 OrigClientOrderId 之一。
CancelAllOpenOrders	交易对	
AutoCancelAllOpenOrders	Symbol, CountdownTimer	在指定倒计时结束时取消指定交易对的所有未结挂单。
QueryCurrentOpenOrder	交易对	
GetOpenOrders	交易对	获取某个交易对的所有未结订单。访问时不指定交易对请谨慎。
GetAllOrders	交易对	获取所有账户订单；包括活跃、已取消或已成交的订单。
GetAccountBalance		
GetAccountInformation		获取当前账户信息。
ChangeInitialLeverage	Symbol, Leverage	修改特定交易对市场的用户初始杠杆。
ChangeMarginType	Symbol, MarginType	
ModifyIsolatedPositionMargin	Symbol, Amount, Type	
GetPositionMarginChangeHistory	交易对	
GetPositionInformation	交易对	
GetAccountTradeList	交易对	
GetIncomeHistory	交易对	
GetNotionalLeverageBracket	交易对	
TestNewOrder	Symbol、Side、PositionSide、Type	测试新订单创建和签名/recvWindow 时长。创建并验证新订单，但不将其发送到撮合引擎。
ModifyOrder	交易对	修改现有订单。
NewBatchOrders	BatchOrders	批量下单。
ModifyBatchOrders	BatchOrders	修改多个订单。
CancelBatchOrders	交易对	批量取消订单。
GetOrderAmendment	交易对	获取订单修改历史。
CountdownCancelAll	Symbol, CountdownTime	在指定倒计时结束时取消指定交易对的所有未结挂单。
GetForceOrders	交易对	获取用户的强制平仓订单。
GetADLQuantile	交易对	获取持仓的 ADL 分位数估计。
GetAccountBalanceV3		获取合约账户余额（V3）。
GetAccountInformationV3		获取当前账户信息（V3）。
GetPositionInformationV3	交易对	获取当前持仓信息（V3）。

GetCommissionRate	交易对	获取用户佣金费率。
GetAccountConfig		获取当前账户配置。
GetSymbolConfig	交易对	获取品种配置。
GetOrderRateLimit		获取用户的订单速率限制。
GetApiTradingStatus	交易对	获取 API 交易量化规则指标。
ChangeMultiAssetsMode	MultiAssetsMargin	更改用户的多资产模式。多资产模式：true；单资产模式：false。
GetMultiAssetsMode		获取用户当前的多资产模式。
SetFeeBurn	FeeBurn	更改用户的 BNB 手续费燃烧状态。
GetFeeBurn		获取用户的 BNB 手续费减免状态。
CreateListenKey		启动新的用户数据流。若未发送 keepalive，流将在 60 分钟后关闭。
KeepAliveListenKey		保活用户数据流以防止超时。
CloseListenKey		关闭用户数据流。

事件

Binance Futures 消息在 TsgcWebSocketClient 组件中接收，您可以使用以下事件：

OnConnect

成功连接到 Binance 服务器后。

OnDisconnect

与 Binance 服务器断开连接后

OnMessage

服务器发送给客户端的消息在此事件中处理。

OnError

如果协议出现任何错误，将调用此事件。

OnException

如果存在未处理的异常，将调用此事件。

此外，在 Binance API 组件中有一个名为 **OnBinanceHttpException** 的特定事件，每次调用 HTTP 请求（REST API 或 WebSocket 用户数据流）出错时触发。

(*) 由于 Binance 服务器的变更，Rad Studio 10.1 之前的 Indy 版本将无法连接到测试服务器。此问题不影响企业版，或已将 Indy 版本升级至最新版本的情况。

API Binance Futures | 交易

Binance 允许您通过其 REST API 进行期货交易。

配置

首先，您必须在 Binance 账户中创建一个 **API Key** 并添加期货交易权限。

完成此操作后，您可以开始进行期货交易。

首先，您必须选择是否要使用 **USDT** 或 **COIN** 期货进行交易，有一个名为 `FuturesContracts` 的属性，您可以在其中设置想要交易的期货合约

然后，在 Binance Futures 客户端组件中设置您的 **ApiKey** 和 **ApiSecret**，这将用于对发送到 Binance 服务器的请求进行签名。

下达订单

要下达新订单，只需调用 Binance Futures 客户端组件的 **REST_API.NewOrder** 方法即可。

根据订单类型（市价、限价等），API 所需的字段多少有所不同。

必填字段

- **Symbol**：产品 ID 符号，示例：BTCUSD_210326
- **方向**：买入或卖出
- **type**：订单类型
 - LIMIT
 - MARKET
 - STOP
 - TAKE_PROFIT
 - STOP_MARKET
 - TAKE_PROFIT_MARKET
 - TRAILING_STOP_MARKET

基于类型的其他必填字段

- **LIMIT**：timeInForce、quantity、price
- **MARKET**：数量
- **STOP/TAKE_PROFIT**：quantity, price, stopPrice
- **STOP_MARKET/TAKE_PROFIT_MARKET**：stopPrice
- **TRAILING_STOP_MARKET**：callbackRate

下订单时，有 2 种可能性：

1. **成功**：函数 `NewOrder` 返回 Binance 服务器发送的消息。
2. **错误**：异常将在 `OnBinanceHTTPException` 事件中返回。

API SocketIO

SocketIO

Socket.IO 是一个用于实时 Web 应用程序的 JavaScript 库。它实现了 Web 客户端和服务端之间实时、双向的通信。它由两部分组成：在浏览器中运行的客户端库，以及用于 Node.js 的服务器端库。两个组件具有几乎相同的 API。与 Node.js 一样，它是事件驱动的。

消息类型

0：打开（当新传输打开时由服务器发送（重新检查））

1：关闭（请求关闭此传输，但不终止连接本身。）

2：ping（由客户端发送。服务器应以包含相同数据的 pong 数据包作答）

示例

客户端发送：2probe

server sends: 3probe

3：pong（由服务器发送以响应 ping 数据包。）

4：字符串消息（实际消息，客户端和服务端应使用数据调用其回调。）

示例：

42/chat,["join",{"room:1}"]

4 是 engine.io 协议中的消息包类型。

2 是 socket.io 协议中的 EVENT 类型

/chat 是由 socket.io 处理的数据

socket.io 将触发 "join" 事件

将传递 "room: 1" 数据。仅当命名空间为 / 时，可以省略命名空间。

5：升级（在 engine.io 切换传输之前，它会测试服务器和客户端是否可以通过该传输进行通信。若测试成功，客户端将发送升级数据包，请求服务器刷新旧传输上的缓存并切换到新传输。）

6：noop（空操作包。主要用于在收到新的 WebSocket 连接时强制触发轮询周期。）

属性

API：指定 SocketIO 版本：

ioAPI0：支持 socket.io 0.* 服务器（默认选中）

ioAPI1：支持 socket.io 1.* 服务器

ioAPI2：支持 socket.io 2.* 服务器

ioAPI3：支持 socket.io 3.* 服务器

ioAPI4：支持 socket.io 4.* 服务器

Base64：若启用，二进制消息将以 base64 格式接收。

HandShakeCustomURL：允许自定义用于获取 socket.io 会话的 URL。

HandShakeTimestamp：仅在需要将时间戳作为参数随新会话请求一同发送时启用此项（若尝试访问 gevent-socketio Python 服务器，请启用此属性）。

HandShakeAuthToken：如果服务器需要身份验证令牌，请在此处设置身份验证令牌。

命名空间：允许在连接到服务器时设置命名空间。

轮询：禁用此属性后，客户端将直接使用 WebSocket 作为传输方式连接到服务器。

Parameters：允许设置连接参数。

EncodeParameters：如果启用，则对参数进行编码。

方法

使用 WriteData 方法向 socket.io 服务器发送消息（请参阅消息类型部分）。

1. 使用 John 作为用户名，调用方法"add user"并传入一个参数。

```
WriteData("42[\"add user\", \"John\"]");
```

事件

OnHTTPRequest

在建立新的 WebSocket 连接之前，socket.io 服务器要求客户端打开一个新的 HTTP 连接以获取新的会话 ID。在某些情况下，socket.io 服务器需要使用 HTTP 标头进行身份验证。您可以使用此事件添加自定义 HTTP 标头，例如 Basic 授权或 Bearer 令牌认证。

OnAfterConnect

socket.io 连接成功且客户端可以向服务器发送消息后调用此事件。例如，您可以在此处订阅命名空间。

OnHTTPConnectionSSL

当 WebSocket 服务器需要安全连接时，若客户端尝试连接该服务器，您可能会收到如下错误消息：

Error connecting with SSL. error:XXXXXXX:SSL routines:ssl3_read_bytes:tlsv1 alert protocol version

此错误表示您的客户端正尝试使用服务器不支持的 TLS 版本进行连接。

要解决此错误，您必须处理 WebSocket 客户端组件的 OnSSLAfterCreateHandler 事件，并设置更新的 TLS 版本。例如：此处我们将 TLS 1.2 设置为协议版本。

```
void OnHTTPConnectionSSL(TObject *Sender; TIdSSLIOHandlerSocketBase *aSSLHandler)
{
    static_cast<tidssliohandlersocketopenssl*>(aSSLHandler)->SSLOptions->Method = sslvTLSv1_2;
}
```


API Coinbase

Coinbase

支持的 API

- [WebSockets API](#)：连接到公共 WebSocket 服务器，提供实时行情数据更新。
- [REST API](#)：REST API 提供账户和订单管理以及公共市场数据的端点。

最常见的用途

- **WebSockets API**
 - [如何连接到 WebSocket API](#)
 - [如何订阅 WebSocket 频道](#)
- **REST API**
 - [如何获取市场数据](#)
 - [如何使用私有 REST API](#)
 - [如何下单](#)
 - [如何使用沙盒账户](#)
 - [私有请求时间](#)

WebSockets API

WebSocket 数据源公开可用，为订单和交易提供实时市场数据更新。生产环境支持两个端点：

- **市场数据**是提供订单和交易更新的数据流。目前大多数频道无需身份验证即可使用。
- **用户订单数据**提供用户订单的更新。

您可以订阅以下频道：

方法	参数	描述
SubscribeHeartBeat		实时服务器 ping，以保持所有连接处于打开状态。
SubscribeStatus	aProductId ：产品的 id	按预设间隔发送所有产品和货币信息
SubscribeCandles	aProductId ：产品的 id	产品蜡烛图实时更新
SubscribeTicker	aProductId ：产品的 id	每次成交时实时价格更新
SubscribeTickerBatch	aProductId ：产品的 id	每 5000 毫秒实时更新一次价格
SubscribeLevel2	aProductId ：产品的 id	所有更新及保持订单簿快照的最简便方式
SubscribeMarketTrades	aProductId ：产品的 id	每次市场成交时的实时更新
SubscribeUser	aProductId ：产品的 id	仅发送包含已认证用户的消息
SubscribeFuturesBalanceSummary		用户合约余额每次变化时的实时更新

User 和 FuturesBalanceSummary 频道需要认证，请先在您的 Coinbase 账户中申请 API 密钥，然后在组件的 Coinbase 属性中设置这些值：

- ApiKey
- ApiSecret

身份验证将带来以下几项好处：

1. 您作为一方参与的消息会被展开，并包含更多有用字段。
2. 您将收到私有消息，例如您设置的止损单的生命周期信息

REST API

私有端点

私有端点可用于订单管理和账户管理。

在签署任何请求之前，您必须通过 Coinbase Pro 网站创建一个 API 密钥。该 API 密钥将绑定到特定的配置文件。创建密钥后，您将获得以下 3 项信息，必须牢记：

- 密钥
- 密钥
- 密码短语

密钥和密钥 Secret 将由 Coinbase Pro 随机生成并提供；密码短语由您提供，以进一步保护您的 API 访问安全。Coinbase Pro 存储您密码短语的加盐哈希值用于验证，但如果您忘记密码短语，将无法找回。

您可以限制 API 密钥的功能。在创建密钥之前，您必须选择该密钥所需的权限。可选权限如下：

- 查看权限，允许读取密钥。包括所有 GET 端点。
- 转账——允许密钥代表账户转移货币，包括存款和提款。请谨慎启用——API 密钥转账将绕过双重身份验证。
- Trade（交易）- 允许密钥下单及获取交易数据，包括 POST /orders 和多个 GET 端点。

账户

方法	Arguments	描述
ListAccounts		从 API 密钥的配置文件中获取交易账户列表。
GetAccount	aAccountId ：账户 ID	单个账户的信息。当您知道 account_id 时使用此端点。API 密钥必须属于与该账户相同的配置文件。

订单

方法	Arguments	描述
PlaceNewOrder	aOrder ：包含订单所有可能字段的类	下新订单。仅在需要访问高级订单选项时使用。
PlaceMarketOrder	aSide ：买入或卖出 aProductId ：产品的 ID。 aQuoteSize ：交易对中第二种资产的数量。 aBaseSize ：交易对中第一种资产的数量 aClient_oid ：您选择用于标识订单的订单 ID	下达新的市价单。
PlaceLimitOrder	aSide ：买入或卖出 aProductId ：产品的 ID。 aQuoteSize ：交易对中第二个资产的数量。	下新限价单。

	aBaseSize ：交易对中第一种资产的数量 aLimitPrice ：限价价格 Client_oid ：您选择用于标识订单的订单 ID	
PlaceStopOrder	aSide ：买入或卖出 ProductId ：产品的 id aBaseSize ：交易对中第一个资产的数量 StopPrice ：止损价格 aLimitPrice ：价格限制 aStopDirection ：止损或入场 Client_oid ：您选择用于标识订单的订单 ID	下达新的止损单
CancelOrder	aOrderId ：订单 ID	取消之前下的订单。订单必须属于 API 密钥所属的账户资料。
EditOrder	aOrderId ：订单 ID aPrice ：价格 aSize ：数量	以指定的新数量或新价格修改订单
EditOrderPreview	aOrderId ：订单 ID aPrice ：价格 aSize ：数量	预览修改订单请求，指定新的数量或新的价格。
ListOrders		获取按可选查询参数（product_id、order_status 等）过滤的订单列表。
GetOrder	aOrderId ：订单 ID	通过订单 ID 获取单个订单。
PreviewOrder		预览订单。
ClosePosition	aOrderId ：订单 ID aProductId ：产品的 id aSize ：数量	下单以平掉指定 product_id 的所有未平仓头寸。

市场数据

方法	Arguments	描述
GetPublicProducts		获取可交易货币对列表。
GetPublicProduct	aProductId ：产品的 id	通过产品 ID 获取单个产品的信息。
GetPublicProductBook	aProductId ：产品的 ID	获取单个产品的买卖报价列表，可使用 limit 参数自定义显示的详细程度。
GetPublicProductCandles	aProductId ：产品的 ID aStart ：时间区间的起始时间 aEnd ：时间区间的结束时间 aGranularity ：每根蜡烛图所代表的时间框架。	按产品 ID 获取单个产品的利率，按时间桶分组。
GetTrades	aProductId ：产品的 ID	按产品 ID 获取最近成交（ticks）及最优买卖价的快照信息。
GetTime		从 Coinbase Advanced API 获取当前时间。

成交明细

方法	Arguments	描述
GetFillsByOrderId		获取按订单 ID 过滤的成交列表
GetFillsByProductId		获取按产品 ID 过滤的成交列表。

GetFillsByTradeId

获取按交易 ID 过滤的成交列表

转换

方法	Arguments	描述
CreateConvertQuote		创建货币之间的兑换报价。
CommitConvertTrade		提交转换交易。
GetConvertTrade		获取兑换交易详情。

费用

方法	Arguments	描述
GetTransactionSummary		获取交易手续费摘要。

产品（已认证）

方法	Arguments	描述
ListProducts		列出可用产品。
GetProduct	aProductId ：产品的 id	获取特定产品。
GetProductBook	aProductId ：产品的 id	获取产品订单簿。
GetProductCandles	aProductId ：产品的 id	获取产品 OHLCV 蜡烛图数据。
GetMarketTrades	aProductId ：产品的 id	获取最近的市场成交记录。
GetBestBidAsk		获取最优买卖价格。

投资组合

方法	Arguments	描述
ListPortfolios		列出所有投资组合。
CreatePortfolio		创建新投资组合。
DeletePortfolio		删除投资组合。
GetPortfolioBreakdown		获取投资组合细分详情。
MovePortfolioFunds		在投资组合之间划转资金。

永续合约

方法	Arguments	描述
GetPerpetualsPortfolioSummary		获取永续合约投资组合摘要。
ListPerpetualsPositions		列出永续仓位。
GetPerpetualsPosition		获取特定的永续合约持仓。

Coinbase | 连接 WebSocket API

要连接到 Coinbase WebSocket API, 只需创建一个新的 Coinbase API 客户端并将其附加到 TsgcWebSocketClient。请参见以下示例：

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
oCoinbase->Client = oClient;
oClient->Active = true;
```

Coinbase | 订阅 WebSocket 频道

Coinbase 提供多种频道，您可以订阅以获取市场数据、订单等的实时更新..... 以下是如何订阅 Ticker 的示例：

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
oCoinbase->Client = oClient;
oCoinbase->SubscribeTicker("ETH-USD");

void OnCoinbaseMessage(TObject *Sender, string aType, string aRawMessage)
{
    // here you will receive the ticker updates
}
```

Coinbase Pro | 获取市场数据

Coinbase 通过 REST 端点提供公开市场数据，当您调用这些端点之一时，您将获得所请求市场数据的快照。市场数据端点不需要身份验证，所有用户均可免费访问。

示例：要获取 BTC-USD 的订单簿快照，请执行以下调用

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase(this);  
ShowMessage(oCoinbase->REST_API->GetPublicProductBook("BTC-USD"));
```

Coinbase Pro | 私有 REST API

Coinbase REST API 提供公开和私有端点。私有端点要求对消息进行签名以提高交易安全性。首先，您必须登录 Coinbase 账户并创建一个新 API，您将获得以下值：

- ApiKey
- ApiSecret

这些字段必须在 Coinbase API 客户端组件的 Coinbase 属性中配置。配置完成后，您可以开始向 Coinbase REST API 发起私有请求

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase(this);
oCoinbase->Coinbase->ApiKey = "<your api key>";
oCoinbase->Coinbase->ApiSecret = "<your api secret>";
ShowMessage(oCoinbase->REST_API->ListAccounts);
```


Coinbase Pro | 私有请求时间

当您向 Coinbase 发送私有请求时，消息会被签名以提高请求的安全性。消息采用本地时间并将其包含在签名消息中，如果本地时间与 Coinbase 服务器时间相差超过 30 秒，请求将被拒绝。因此，验证您的本地时间是否已同步非常重要，您可以通过适用于您操作系统的时间同步方法来完成此操作。

您可以调用 **GetTime** 方法来检查 Coinbase Pro 服务器时间，该方法将返回 Coinbase Pro 服务器的时间

Coinbase Pro | 下单

要在 Coinbase 中下新订单，首先需要获取您的 API 密钥以访问私有数据。请参阅以下文章：[如何使用私有 REST API](#)。

配置好 API 密钥后，即可开始下单

市价单

下达新的市价单，买入 0.002 份 BTC-USD 合约

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase(this);
oCoinbase->Coinbase->ApiKey = "your api key";
oCoinbase->Coinbase->ApiSecret = "your api secret";
oCoinbase->Coinbase->ApiPassphrase = "your passphrase";
ShowMessage(oCoinbase->REST_API>PlaceMarketOrder(coisBuy, "BTC-USD", 0.002
, 0
));
```

限价单

下一个新的限价订单，以 10000 的价格限制买入 0.002 份 BTC-USD 合约

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase(this);
oCoinbase->Coinbase->ApiKey = "your api key";
oCoinbase->Coinbase->ApiSecret = "your api secret";
oCoinbase->Coinbase->ApiPassphrase = "your passphrase";
ShowMessage(oCoinbase->REST_API>PlaceLimitOrder(coisBuy, "BTC-USD", 0.002,
0,
10000));
```

Coinbase Pro 沙盒账户

Coinbase 允许您使用沙盒账户进行无真实资金的交易。该账户需要创建与生产账户不同的 API 密钥。要使用沙盒账户，只需在向 REST API 发起任何请求之前，将 **Coinbase.SandBox** 属性设置为 **true**。

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase(this);
oCoinbase->Coinbase->ApiKey = "your api key";
oCoinbase->Coinbase->ApiSecret = "your api secret";
oCoinbase->Coinbase->SandBox = true;
ShowMessage(oCoinbase->REST_API>ListAccounts);
```

API SignalRCore

SignalRCore

ASP.NET Core SignalR 是一个开源库，可简化向应用程序添加实时 Web 功能的过程。实时 Web 功能使服务器端代码能够即时向客户端推送内容。

SignalR 的良好候选场景：

- 需要服务器高频更新的应用程序。例如游戏、社交网络、投票、拍卖、地图和 GPS 应用程序。
- 仪表板和监控应用。示例包括公司仪表板、即时销售更新或旅行提醒。
- 协作应用。白板应用和团队会议软件是协作应用的例子。
- 需要通知的应用程序。社交网络、电子邮件、聊天、游戏、旅行提醒以及许多其他应用程序使用通知。

SignalRCore sgcWebSockets 组件使用 WebSocket 作为传输方式连接到 SignalRCore 服务器，如果此传输方式不受支持，将引发错误。

Hubs

SignalRCore 使用集线器（Hub）在客户端与服务器之间进行通信。SignalRCore 提供两种集线器协议：基于 JSON 的文本协议和基于 MessagePack 的二进制协议。sgcWebSockets 组件仅实现了 JSON 文本协议，用于与 SignalRCore 服务器通信。

要配置 Hub 客户端将使用哪个 Hub，只需在客户端连接到服务器之前，在 **SignalRCore/Hub** 属性中设置 Hub 的名称。

连接

当客户端向服务器打开新连接时，会发送包含格式协议和版本的请求消息。sgcWebSockets 始终以 JSON 格式发送协议。若服务器不支持该协议，将回复错误，可通过 **OnSignalRCoreError** 事件处理；若连接成功，则调用 **OnSignalRCoreConnect** 事件。

当客户端连接到 SignalRCore 服务器时，可以发送 **ConnectionId** 在会话间标识客户端。要获取新的连接 ID，只需正常连接到服务器并通过 **OnBeforeConnectEvent** 获知 ConnectionId。若要重连并传递之前的连接 ID，请使用 **ReConnect** 方法并将 ConnectionId 作为参数传入。

SignalRCore 协议

SignalR 协议是一种基于消息传输的双向 RPC 协议。连接的任意一方均可调用另一方的过程，过程可返回零个或多个结果或一个错误。示例：客户端可以请求服务器的方法，服务器也可以请求客户端的方法。服务器与客户端之间交换以下消息：

- **HandshakeRequest**：客户端发送给服务器，用于协商消息格式。
- **HandshakeResponse**：服务器向客户端回复对之前 HandshakeRequest 消息的确认。如果握手失败，则包含错误。
- **Close**：当连接关闭时由客户端或服务器调用。如果连接因错误关闭，则包含错误信息。
- **调用**：客户端或服务器向另一个对等方发送消息，以调用带参数或不带参数的方法。
- **StreamInvocation**：客户端或服务器向另一对端发送消息，以调用带参数或不带参数的流式方法。响应将被拆分为不同的项目。
- **StreamItem**：是来自先前 StreamInvocation 的响应。
- **Completion**：表示上一次调用或 StreamInvocation 已完成。如果处理成功则包含结果，如果出错则包含错误信息。
- **CancellInvocation**：取消之前的 StreamInvocation 请求。
- **Ping**：用于检查连接是否仍然存活的消息。

SignalRCore 编码

SignalRCore 允许使用以下编码方式：

- **JSON**：目前唯一支持的编码。
- **MessagePack**

目前仅支持 JSON，但可以使用外部 MessagePack 库对发送的消息进行编码，从而使用 MessagePack。有关更多信息，请参阅下方的 MessagePack 章节。

编码协议的配置在 SignalRCore.Protocol 属性中定义。默认值为 **srcpJSON**。

授权

可以启用身份验证，将用户与每个连接关联，并过滤哪些用户可以访问资源。身份验证使用 Bearer 令牌实现：客户端提供访问令牌，服务器验证此令牌并使用它来识别用户。

在标准 Web API 中，承载令牌通过 HTTP 标头发送，但使用 WebSocket 时，令牌作为查询字符串参数传输。支持以下方法：

srcRequestToken

若启用了身份验证，流程如下：

1. 首先尝试从服务器获取有效令牌。打开到 Authentication.RequestToken.URL 的 HTTP 连接，并使用用户名和密码数据进行 POST。
2. 如果前一步成功，则返回令牌；否则返回错误。
3. 若返回令牌，则打开新的 HTTP 连接进行协商。此时令牌作为 HTTP 头传递。
4. 如果上一步成功，打开 WebSocket 连接并将令牌作为查询字符串参数传递。

- **Authentication.Enabled**：若激活，则在建立 WebSocket 连接之前需进行授权。
- **Authentication.Username**：提供给服务器用于身份验证的用户名。
- **Authentication.Password**：提供给服务器以进行身份验证的密码。
- **Authentication.RequestToken.PostFieldUsername**：用于传输用户名的字段名称（取决于配置，请查看 HTTP JavaScript 页面以了解使用的名称）。
- **Authentication.RequestToken.PostFieldPassword**：传输密码的字段名称（取决于配置，请查看 http javascript 页面以了解所用名称）。
- **Authentication.RequestToken.URL**：请求令牌的 URL。
- **Authentication.RequestToken.QueryFieldToken**：WebSocket 连接中使用的查询字符串参数名称。

srcSetToken

在此处，您直接将令牌传递给 SignalRCore 服务器（因为令牌是从另一台服务器获取的）。

- **Authentication.Enabled**：如果激活，在建立 WebSocket 连接之前将使用授权。
- **Authentication.SetToken.Token**：获取到的令牌值。

访问令牌可以作为查询参数发送（这是默认选项），也可以作为 Bearer Token 在 HTTP 头中发送。使用属性 Authentication.TokenParam 配置此行为。

- **srctQuery**：access_token 通过 WebSocket 连接的查询 URL 传递。
- **srctHeader**：access_token 作为 Bearer Token 通过 HTTP 头传递。

srcBasic

此选项使用**基本身份验证**，此身份验证方法需要配置 SignalRCore 组件和 [TsgcWebSocketClient](#)。

示例：如果服务器需要基本身份验证，用户名为"user"，密码为"secret"，请按如下所示配置组件。

```
// websocket client
TsgcWebSocketClient* WSClient = new TsgcWebSocketClient();
WSClient->Authentication->Enabled = true;
WSClient->Authentication->Basic->Enabled = true;
WSClient->Authentication->URL->Enabled = false;
WSClient->Authentication->Session->Enabled = false;
WSClient->Authentication->Token->Enabled = false;
WSClient->Authentication->User = "user";
WSClient->Authentication->Password = "secret";
// signalr core
TsgcWSAPI_SignalRCore* Signal = new TsgcWSAPI_SignalRCore();
Signal->SignalRCore->Authentication->Enabled = true;
Signal->SignalRCore->Authentication->Authentication = srcaBasic;
Signal->SignalRCore->Authentication->Username = "user";
Signal->SignalRCore->Authentication->Password = "secret";
Signal->Client = WSClient;
```

客户端与服务端之间的通信

服务器与客户端之间存在三种交互类型：

调用

调用方向被调用方发送消息，并期待一条表明调用已完成的消息，以及调用结果（可选）

示例：客户端调用 `SendMessage` 方法，并将用户名和文本消息作为参数传递。发送调用 ID 以从服务器获取结果消息。

```
SignalRCore->Invoke("SendMessage", ARRAYOFCONST(("John", "Hello All.")), "id-000001");

void OnSignalRCoreCompletion(TObject *Sender, TSignalRCore_Completion *Completion)
{
    if (Completion->Error != "")
    {
        ShowMessage("Something goes wrong.");
    }
    else
    {
        ShowMessage("Invocation Successful!");
    }
}
```

非阻塞调用

调用方向被调用方发送一条消息，且不期望收到此次调用的任何后续消息。调用时可不带调用 ID 值，这表明该调用为“非阻塞”。

示例：客户端调用 `SendMessage` 方法，并将用户名和文本消息作为参数传入。客户端不期望服务器返回任何关于调用结果的响应。

```
SignalRCore->Invoke("SendMessage", ARRAYOFCONST(("John", "Hello All.")));
```

流式调用

调用方向被调用方发送消息，并期望被调用方返回一个或多个结果，最后跟随一条表示调用结束的消息。

示例：客户端调用 `Counter` 方法，请求以 500 毫秒的间隔返回 10 个数字。

```
SignalRCore->InvokeStream("Counter", [10, 500], "id-000002");

void OnSignalRCoreStreamItem(TObject *Sender, TSignalRCore_StreamItem *StreamItem, bool &Cancel)
```

```

{
    DoLog("#stream item: " + StreamItem->Item);
}

void OnSignalRCoreCompletion(TObject *Sender, TSignalRCore_Completion *Completion)
{
    if (Completion->Error != "")
    {
        ShowMessage("Something goes wrong.");
    }
    else
    {
        ShowMessage("Invocation Successful!");
    }
}
}

```

调用

为了执行单次调用，调用方遵循以下基本流程：

```

void Invoke(const string aTarget, const Array of Const aArguments, const string aInvocationId)
void InvokeStream(const string aTarget, const Array of Const aArguments, const String aInvocationId)

```

为调用分配唯一的 Invocation ID 值（由调用方选择的任意字符串）。调用包含目标对象、参数和 InvocationId 的 Invoke 或 InvokeStream 方法（如果不发送 InvocationId，则不会收到完成结果）。

如果调用被标记为非阻塞（参见下方“非阻塞调用”），则在此处停止并立即将控制权交还给应用程序。使用匹配的调用 ID 处理 StreamItem 或 Completion 消息。

```

SignalRCore->InvokeStream("Counter", [10, 500], "id-000002");

void OnSignalRCoreStreamItem(TObject *Sender, TSignalRCore_StreamItem *StreamItem, bool &Cancel)
{
    if (StreamItem->InvocationId == "id-000002")
    {
        DoLog("#stream item: " + StreamItem->Item);
    }
}

void OnSignalRCoreCompletion(TObject *Sender, TSignalRCore_Completion *Completion)
{
    if (Completion->InvocationId == "id-000002")
    {
        if (Completion->Error != "")
        {
            ShowMessage("Something goes wrong.");
        }
        else
        {
            ShowMessage("Invocation Successful!");
        }
    }
}
}

```

您可以进行单次调用并等待完成。

```

bool InvokeAndWait(const String aTarget, System::TVarRec *aArguments, string aInvocationId,
    TSignalRCore_Completion &Completion, const int aTimeout = 10000);
bool InvokeStreamAndWait(const String aTarget, System::TVarRec *aArguments, string aInvocationId,
    TSignalRCore_Completion &Completion, const int aTimeout = 10000);

```

为调用分配唯一的调用 ID 值（任意字符串，由调用方选定）。调用 InvokeAndWait 或 InvokeStreamAndWait 方法，传入被调用的目标、参数和调用 ID。程序将等待直至完成事件被触发或超时。

```

{
    TSignalRCore_Completion oCompletion;
    if (SignalRCore->InvokeStreamAndWait("Counter", ARRAYOFCONST((10, 500)), "id-000002", oCompletion))
    {
        DoLog("#invoke stream ok: " + oCompletion->Result);
    }
    else
    {
        DoLog("#invocke stream error: " + oCompletion->Error);
    }
}

```

```

    }
}

void OnSignalRCoreStreamItem(TObject *Sender, TSignalRCore_StreamItem *StreamItem, bool &Cancel)
{
    if (StreamItem->InvocationId == "id-000002")
    {
        DoLog("#stream item: " + StreamItem->Item);
    }
}

```

取消调用

如果客户端希望在服务器发送 Completion 消息之前停止接收 StreamItem 消息，客户端可以发送与启动流的 StreamInvocation 消息使用相同 InvocationId 的 CancellInvocation 消息。

```

void OnSignalRCoreStreamItem(TObject *Sender, TSignalRCore_StreamItem *StreamItem, bool &Cancel)
{
    if (StreamItem->InvocationId == "id-000002")
    {
        Cancel = true;
    }
}

```

客户端结果

只有收到完成消息时，调用才被视为完成。如果客户端从服务器接收到调用，将触发 OnSignalRCoreInvocation 事件。

```

void OnSignalRCoreInvocation(TObject *Sender, TSignalRCore_Invocation *Invocation)
{
    if (Invocation->Target == "SendMessage")
    {
        ... your code here ...
    }
}

// Once invocation is completed, call Completion method to inform server invocation is finished.
// If result is successful, then call CompletionResult method:
SignalRCore->CompletionResult("id-000002", "ok");

// If not, then call CompletionError method:
SignalRCore->CompletionError("id-000002", "Error processing invocation.");

```

关闭连接

由客户端在连接关闭时发送。如果连接因错误而关闭，则包含错误原因。

```

SignalRCore->Close("Unexpected message").

// If the server close connection by any reason, OnSignalRCoreClose event will be called.
void OnSignalRCoreClose(TObject *Sender, TSignalRCore_Close *Close)
{
    DoLog("#closed: " + Close->Error);
}

```

Ping

SignalR Hub 协议支持“保活”消息，用于确保底层传输连接保持活跃。这些消息有助于确保：

在空闲期间（消息发送较少时），代理不会关闭底层连接。如果底层连接在未正常终止的情况下断开，应用程序将尽快得到通知。

保活行为通过调用 Ping 方法或在 WebSocket 客户端上启用 HeartBeat 来实现。如果服务器向客户端发送 Ping，客户端将自动发送响应，并调用 OnSignalRCoreKeepAlive 事件。

```
void OnSignalRCoreKeepAlive(TObject *Sender)
{
    DoLog("#keepalive");
}
```

MessagePack

在 SignalR 协议的 MsgPack 编码中，每条消息表示为一个单独的 MsgPack 数组，其中包含与给定 hub 协议消息属性对应的项。数组项可以是基本值、数组（例如方法参数）或对象（例如参数值）。数组中的第一项是消息类型。

请参阅 [MessagePack 文档](#)，了解如何对发送的消息进行编码。

每次收到新消息时，都会在 OnSignalRCoreMessagePack 事件中分发。消息内容可通过读取 Data Stream 参数获取。JSON 参数默认为空；如果您将 MessagePack 消息转换为 JSON，组件将像处理 JSON 编码消息一样处理该 JSON（因此 OnSignalRCoreCompletion、OnSignalRCoreInvocation 等事件将被触发）。

API SignalR

SignalR

SignalR 组件使用 WebSocket 作为传输方式连接到 SignalR 服务器，若不支持此传输方式，将引发错误。SignalR 客户端组件有一个名为 SignalR 的属性，您可以在其中设置以下数据：

- **Hubs**：包含客户端正在订阅的 Hub 列表。
- **ProtocolVersion**：客户端使用的协议版本，支持 1.2 到 1.5 的协议版本。
- **UserAgent**：用于连接到 SignalR 服务器的用户代理。

客户端支持发送文本或二进制数据。

Hubs 消息

Hubs API 使从客户端调用服务器方法以及从服务器调用客户端方法成为可能。用于持久连接的协议功能不够丰富，无法直接表达 RPC（远程过程调用）语义。但这并不意味着用于 Hub 连接的协议与用于持久连接的协议完全不同。事实上，Hub 连接使用的协议主要是持久连接协议的扩展。

当客户端调用服务器方法时，不再像持久连接那样发送自由格式的字符串，而是发送一个包含调用该方法所需所有信息的 JSON 字符串。以下是客户端调用服务器方法时发送的示例消息：

```
WriteData("{\"H\":\"chathub\",\"M\":\"Send\",\"A\":[\"CBuilder Client\",\"Test message\"],\"I\":0}
```

载荷具有以下属性：

I——调用标识符——用于将响应与请求进行关联

H – hub 的名称

M — 方法名称

A – 参数（一个数组，如果方法没有任何参数则可以为空）

如果字符串参数包含双引号，请将 " 替换为 \"

示例：如果参数为 {"test":1}，请将参数发送为 {"test\\":1}

```
WriteData("{\"H\":\"chathub\",\"M\":\"Send\",\"A\":[\"test\\\":1],\"I\":0}");
```

授权

可以启用身份验证，将用户与每个连接关联，并过滤哪些用户可以访问资源。身份验证使用 Bearer 令牌实现：客户端提供访问令牌，服务器验证此令牌并使用它来识别用户。

目前仅支持 Bearer Token：

在此，您将令牌直接传递给 Signal 服务器（因为令牌已从另一台服务器获取）。

- **Authentication.Enabled**：如果激活，在建立 WebSocket 连接之前将使用授权。
- **Authentication.Authentication**：支持 2 种认证类型：Bearer 令牌或 Cookie。两者均需要外部方式来获取所需的值。
 - **BearerToken**：获取的令牌值。
 - **Cookie**：设置所需 Cookie 的值。

```
TsgcWSAPI_Signal *oSignalR = new TsgcWSAPI_Signal(NULL);
oSignalR->SignalR->Enabled = true;
oSignalR->SignalR->Authentication = srcBearerToken;
oSignalR->SignalR->BearerToken->Token = "token here";
```

该组件具有以下事件：

OnSignalRConnect

当客户端成功连接到服务器时触发此事件。

OnSignalRDisconnect

当客户端从服务器断开时触发此事件。

OnSignalRError

当 WebSocket 连接发生错误时调用此事件。

OnSignalRMessage

用于持久连接的协议非常简单。发送到服务器的消息只是原始字符串，没有特定的格式要求。发送给客户端的消息则更有结构。您可以在消息中找到以下属性：

C — 消息 ID，所有非 KeepAlive 消息均存在
M — 包含实际数据的数组。

```
{"C": "d-9B7A6976-B, 2|C, 2", "M": ["Welcome!"]}
```

OnSignalRBinary

当从服务器收到二进制数据时，将调用此事件。

OnSignalRResult

当服务器方法被调用时，服务器会返回调用已完成的确认，向客户端发送调用 id，以及如果方法有返回值则发送该返回值；如果调用方法失败则发送错误。

以下是服务器方法调用的示例结果：

```
{"I": "0"}
```

调用标识符为 "0" 的服务器 void 方法已成功完成。

```
{"I": "0", "R": 42}
```

调用标识符为 "0" 的服务器方法成功完成并返回值 42。

```
{"I": "0", "E": "Error occurred"}
```

OnSignalRKeepAlive

当从服务器收到 KeepAlive 消息时触发此事件。

API Kraken

Kraken

概述

WebSockets API 提供实时市场数据更新。WebSockets 是一种双向协议，可提供最快的实时数据，帮助您构建实时应用程序。以下公共消息类型无需身份验证。私有数据消息可以在单独的已验证端点上订阅。

Kraken 也提供 REST API，包括公开市场数据和私有用户数据（需要身份验证）。

配置

Private API 需要从您的 Kraken 账户创建 API 密钥。

Kraken 允许在 WebSocket 协议上使用测试环境，启用 Kraken 属性中的 Beta 属性即可使用此测试功能。

支持的 API

- [WebSockets 公共 API](#)：连接到公共 WebSocket 服务器。
- [WebSockets 私有 API](#)：连接到私有 WebSocket 服务器，需要 API Key 和 API Secret 进行服务器身份验证。
- [REST 公共 API](#)：连接到公共 REST 服务器。
- [REST 私有 API](#)：连接到公共 REST 服务器，需要 API Key 和 API Secret 对服务器进行身份验证。

Kraken 示例

如何连接到公共 WebSocket 服务器

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Kraken *oKraken = new TsgcWSAPI_Kraken(NULL);
oKraken->Client = oClient;
oClient->Active = true;
```

如何连接到私有 WebSocket 服务器

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Kraken *oKraken = new TsgcWSAPI_Kraken(NULL);
oKraken->Kraken->ApiKey = "your api key";
oKraken->Kraken->ApiSecret = "your api secret";
oKraken->Client = oClient;
oClient->Active = true;
```

如何从 REST API 获取行情

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Kraken *oKraken = new TsgcWSAPI_Kraken(NULL);
oKraken->Client = oClient;
ShowMessage(oKraken->GetTicker(ARRAYOFCNST(("XBTUSD"))));
```

REST API 方法

公共端点

方法	参数	描述
GetSystemStatus		获取当前系统状态。

私有端点

方法	参数	描述
GetExtendedBalance		获取扩展余额信息。
AmendOrder		修改现有订单。
CancelAllOrders		取消所有未成交订单。
CancelAllOrdersAfter		死人开关 — 超时后取消所有订单。
EditOrder		编辑现有订单。
AddOrderBatch		批量添加多个订单。
CancelOrderBatch		批量取消多个订单。
GetWithdrawalMethods		获取可用的提现方式。
GetWithdrawalAddresses		获取提款地址。

如何通过 REST API 获取账户余额

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Kraken *oKraken = new TsgcWSAPI_Kraken(NULL);
oKraken->Kraken->ApiKey = "your api key";
oKraken->Kraken->ApiSecret = "your api secret";
oKraken->Client = oClient;
ShowMessage(oKraken->GetAccountBalance());
```

API Kraken | WebSockets 公共 API

连接

URL: wss://ws.kraken.com (v1) 或 wss://ws.kraken.com/v2 (v2, 推荐)

该组件现在通过 **Version** 属性支持 WebSocket API v2 (默认值为 2)。将 `Kraken.Version := 1` 设置为使用旧版 v1 端点。

套接字打开后，您可以通过发送订阅请求消息来订阅公共频道。

一般注意事项

- 通过 WebSocket 发送和接收的所有消息均以 JSON 格式编码
- 所有浮点字段（包括时间戳）均加引号以保留精度。
- 每个可交易货币对的格式为 A/B，其中 A 和 B 对于标准化资产使用 ISO 4217-A3，对于非标准化的热门资产使用独特符号。
- 时间戳不应视为唯一标识符，也不应视为交易 ID 的别名。此外，时间戳的粒度并不能代表交易速率。

支持的交易对

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

方法

Ping

客户端可向服务器发送 ping 以确认连接是否存活，服务器以 pong 响应。这是应用层 ping，与 WebSocket 标准中由服务器发起的默认 ping 不同。

行情

Ticker 信息包含给定货币对的最优买价和卖价、24 小时成交量、最新成交价、成交量加权平均价等。每次有成交或一批成交发生时，都会发布一条 Ticker 消息。调用 `SubscribeTicker` 方法订阅行情数据：

```
SubscribeTicker([L"XBT/USD"]);
```

如果订阅成功，将调用 **OnKrakenSubscribed** 事件：

```
void OnKrakenSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription, string ChannelName, int ReqID)
{
    DoLog("#subscribed: " + Subscription + " " + Pair + " " + ChannelName);
}
```

调用 `UnSubscribeTicker` 方法取消订阅：

```
UnSubscribeTicker(ARRAYOFCONST((L"XBT/USD")));
```

若取消订阅成功，将调用 `OnKrakenUnSubscribed` 事件：

```
void OnKrakenUnSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
int ReqID)
{
    DoLog("#unsubscribed: " + Subscription + " " + Pair);
}
```

如果订阅/取消订阅时发生错误，将调用 `OnKrakenSubscriptionError` 事件。

```
void OnKrakenSubscriptionError(TObject *Sender, string ErrorMessage, string Pair, string Subscription,
int ReqID)
{
    DoLog("#subscription error: " + ErrorMessage);
}
```

行情更新将通过 `OnKrakenData` 事件通知。

```
[
    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

订阅 OHLC 后，将首先发送最后一根有效蜡烛的快照（不考虑结束时间），然后推送当前正在运行的蜡烛的更新。例如，如果订阅了 1 分钟蜡烛，而 5 分钟内没有成交，则会发布 5 分钟前最后一根 1 分钟蜡烛的快照。可以使用结束时间来判断该蜡烛是否为旧蜡烛。

调用 `SubscribeOHLC` 方法订阅 OHLC，需传入交易对和时间间隔。


```
SubscribeOHLC(ARRAYOFCONST((L"XBT/USD")), kin1min);
```

如果订阅成功，将调用 OnKrakenSubscribed 事件：

```
void OnKrakenSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription, string ChannelName,
    int ReqID)
{
    DoLog("#subscribed: " + Subscription + " " + Pair + " " + ChannelName);
}
```

调用 UnSubscribeOHLC 方法取消订阅：

```
UnSubscribeOHLC(ARRAYOFCONST(("XBT/USD")), kin1min);
```

若取消订阅成功，将调用 OnKrakenUnSubscribed 事件：

```
void OnKrakenUnSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
    int ReqID)
{
    DoLog("#unsubscribed: " + Subscription + " " + Pair);
}
```

如果订阅/取消订阅时发生错误，将调用 OnKrakenSubscriptionError 事件。

```
void OnKrakenSubscriptionError(TObject *Sender, string ErrorMessage, string Pair, string Subscription,
    int ReqID)
{
    DoLog("#subscription error: " + ErrorMessage);
}
```

OHLC 更新将在 OnKrakenData 事件中通知。

```
[
    42,
    [
        "1542057314.748456",
        "1542057360.435743",
        "3586.70000",
        "3586.70000",
        "3586.60000",
        "3586.60000",
        "3586.68894",
        "0.03373000",
        2
    ],
    "ohlc-5",
    "XBT/USD"
]
```

交易

货币对的交易数据流。

调用 SubscribeTrade 方法订阅交易数据源。

```
SubscribeTrade(ARRAYOFCONST((L"XBT/USD")));
```

如果订阅成功，将调用 OnKrakenSubscribed 事件：

```
void OnKrakenSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
    string ChannelName, int ReqID)
{
    DoLog("#subscribed: " + Subscription + " " + Pair + " " + ChannelName);
}
```

调用 UnSubscribeTrade 方法取消订阅：

```
UnSubscribeTrade(ARRAYOFCONST((L"XBT/USD")));
```

若取消订阅成功，将调用 OnKrakenUnSubscribed 事件：

```
void OnKrakenUnSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
    int ReqID)
{
    DoLog("#unsubscribed: " + Subscription + " " + Pair);
}
```

如果订阅/取消订阅时发生错误，将调用 OnKrakenSubscriptionError 事件。

```
void OnKrakenSubscriptionError(TObject *Sender, string ErrorMessage, string Pair, string Subscription,
    int ReqID)
{
    DoLog("#subscription error: " + ErrorMessage);
}
```

成交更新将在 OnKrakenData 事件中通知。

```
[
  0,
  [
    [
      "5541.20000",
      "0.15850568",
      "1534614057.321597",
      "s",
      "l",
      ""
    ],
    [
      "6060.00000",
      "0.02455000",
      "1534614057.324998",
      "b",
      "l",
      ""
    ]
  ],
  "trade",
  "XBT/USD"
]
```

订单簿

订单簿层级。订阅时，将在指定深度发布一个快照，快照之后将发布层级更新。
调用 SubscribeBook 方法订阅 Book，需传入货币对和深度。

```
SubscribeBook(ARRAYOFCONST((L"XBT/USD")), kde10);
```

如果订阅成功，将调用 OnKrakenSubscribed 事件：

```
void OnKrakenSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription, string ChannelName,
    int ReqID)
{
    DoLog("#subscribed: " + Subscription + " " + Pair + " " + ChannelName);
}
```

调用 UnSubscribeBook 方法取消订阅：

```
UnSubscribeBook(ARRAYOFCONST((L"XBT/USD")), kde10);
```

若取消订阅成功，将调用 OnKrakenUnSubscribed 事件：

```
void OnKrakenUnSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
    int ReqID)
```

```
{
  DoLog("#unsubscribed: " + Subscription + " " + Pair);
}
```

如果订阅/取消订阅时发生错误，将调用 OnKrakenSubscriptionError 事件。

```
void OnKrakenSubscriptionError(TObject *Sender, string ErrorMessage, string Pair, string Subscription,
  int ReqID)
{
  DoLog("#subscription error: " + ErrorMessage);
}
```

书单更新将通过 OnKrakenData 事件通知。

```
[
  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"
]
```

价差

展价数据流，显示已订阅资产对的最佳买卖价。买量和卖量也是消息的一部分。调用 SubscribeSpread 方法订阅展价数据流。

```
SubscribeSpread(ARRAYOFCNST(("XBT/USD")));
```

如果订阅成功，将调用 OnKrakenSubscribed 事件：

```
void OnKrakenSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription, string ChannelName,
  int ReqID)
{
  DoLog("#subscribed: " + Subscription + " " + Pair + " " + ChannelName);
}
```

调用 UnSubscribeSpread 方法取消订阅：

```
UnSubscribeSpread(ARRAYOFCONST((L"XBT/USD")));
```

若取消订阅成功，将调用 OnKrakenUnSubscribed 事件：

```
void OnKrakenUnSubscribed(TObject *Sender, int ChannelId, string Pair, string Subscription,
    int ReqID)
{
    DoLog("#unsubscribed: " + Subscription + " " + Pair);
}
```

如果订阅/取消订阅时发生错误，将调用 OnKrakenSubscriptionError 事件。

```
void OnKrakenSubscriptionError(TObject *Sender, string ErrorMessage, string Pair, string Subscription,
    int ReqID)
{
    DoLog("#subscription error: " + ErrorMessage);
}
```

价差更新将通过 OnKrakenData 事件通知。

```
[
    0,
    [
        "5698.40000",
        "5700.00000",
        "1542057299.545897",
        "1.01234567",
        "0.98765432"
    ],
    "spread",
    "XBT/USD"
]
```

其他方法

您可以通过一个方法订阅/取消订阅所有频道：

```
SubscribeAll(ARRAYOFCONST((L"XBT/USD")));
UnSubscribeAll(ARRAYOFCONST((L"XBT/USD")));
```

如果所有已订阅频道均已订阅，OHLC 间隔值为 1。

事件

OnConnect：当 WebSocket 客户端连接到服务器时。

OnKrakenConnect：在 WebSocket 成功连接且服务器发送系统状态后调用。

OnKrakenSystemStatus：当系统状态变化时调用。

OnKrakenSubscribed：成功订阅频道后调用。

OnKrakenUnSubscribed：在成功取消订阅某个频道后调用。

OnKrakenSubscriptionError：在订阅/取消订阅出错时调用。

OnKrakenData：每次频道订阅有更新时调用。

API Kraken | WebSockets 私有 API

连接

URL: wss://ws-auth.kraken.com (v1) 或 wss://ws-auth.kraken.com/v2 (v2, 推荐)
该组件现在通过 **Version** 属性（默认值：2）支持 WebSocket API v2。

套接字打开后，您可以通过发送经过身份验证的订阅请求消息来订阅私有数据频道。

身份验证

API 客户端必须通过以下 REST API 端点"GetWebSocketsToken"请求认证"令牌"，才能连接到 WebSockets 私有端点。该令牌应在创建后 15 分钟内使用。一旦与 WebSockets API 私有消息（openOrders 或 ownTrades）保持连接，令牌不会过期。

要获取 WebSockets 令牌，必须在 Kraken Options 组件中设置 API Key 和 API Secret，即您账户中 Kraken 提供的 API 密钥

```
Kraken->ApiKey = "api key";
Kraken->ApiSecret = "api secret";
```

方法

OwnTrades

获取自有交易列表，首次订阅时获取最新 50 笔交易列表

```
SubscribeOwnTrades();
```

之后，您可以调用 UnSubscribeOwnTrades 方法取消订阅 OwnTrades

```
UnSubscribeOwnTrades();
```

服务器的响应示例

```
[
  [
    {
      "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"
]
```

未结订单

显示属于已验证 API 密钥的用户的所有未结订单的数据流。初始快照将提供所有未结订单的列表，然后将发送对未结订单列表的任何更新。对于状态变更更新（例如“已关闭”），有效载荷中将包含 `orderid` 和 `status` 字段

```
SubscribeOpenOrders();
```

之后，您可以通过调用 `UnSubscribeOpenOrders` 方法取消订阅 `OpenOrders`。

```
UnSubscribeOpenOrders();
```

服务器的响应示例

```
[
  [
    {
      "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"
]
```

添加订单

向 Kraken 发送新订单

```
TsgcWSKrakenOrder *oKrakenOrder = new TsgcWSKrakenOrder();
oKrakenOrder->Pair = "XBT/USD";
oKrakenOrder->_Type = kosBuy;
oKrakenOrder->OrderType = kotMarket;
oKrakenOrder->Volume = 1;
AddOrder(oKrakenOrder);
```

订单参数列表

```
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):
    vqic = 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

```

服务器的响应示例

```

{
  "descr": "buy 0.01770000 XBTUSD @ limit 4000",
  "event": "addOrderStatus",
  "status": "ok",
  "txid": "ONPNXH-KMKMU-F4MR5V"
}

```

取消订单

取消订单

```
CancelOrder("Order Id");
```

服务器的响应示例

```

{
  "event": "cancelOrderStatus",
  "status": "ok"
}

```

API Kraken | REST 公共 API

连接

URL: <https://api.kraken.com>

Kraken 公开 API 无需任何身份验证。

配置

唯一的配置是是否为 REST HTTP 请求启用日志。如果您想将所有 HTTP 请求/响应保存到文本文件日志中，请启用 HTTPLogOptions。

事件

OnKrakenHTTPException: 当通过 REST API 执行 HTTP 请求时出现任何异常，将调用此事件。

方法

GetServerTime

此方法用于近似估算服务器与客户端之间的时间偏差。以 Unix 格式返回时间。

```
{"error":[],"result":{"unixtime":1586705546,"rfc1123":"Sun, 12 Apr 20 15:32:26 +0000"}}
```

GetAssets

返回有关资产的信息

```
{"error":[],"result":{"ADA":{"aclass":"currency","altname":"ADA","decimals":8,"display_decimals":6}}}}
```

GetAssetPairs

返回一对资产的信息

```
Kraken->REST_API->GetAssetPairs(ARRAYOFCONST((L"XBTUSD")));
```

GetTicker

返回行情信息

```
Kraken->REST_API->GetTicker(ARRAYOFCONST((L"XBTUSD")));
```

GetOHLC

返回开盘-最高-最低-收盘数据。


```
Kraken->REST_API->GetOHLC("XBTUSD");
```

GetOrderBook

返回交易对名称和市场深度的数组。

```
Kraken->REST_API->GetOrderBook("XBTUSD");
```

GetTrades

返回某一交易对的最新交易数据。

```
Kraken->REST_API->GetTrades("XBTUSD");
```

GetSpread

返回某交易对的最近价差数据。

```
Kraken->REST_API->GetSpread("XBTUSD");
```

API Kraken | REST 私有 API

连接

URL: <https://api.kraken.com>

身份验证

REST 私有 API 需要 API Key 和 API Secret，这些值由 Kraken 在您的账户中提供。

```
Kraken->ApiKey = "api key";  
Kraken->ApiSecret = "api secret";
```

方法

GetAccountBalance

返回您的账户余额。

```
Kraken->REST_API->GetAccountBalance();
```

GetTradeBalance

返回您的交易信息。

```
Kraken->REST_API->GetTradeBalance();
```

GetOpenOrders

返回未结订单列表。

```
Kraken->REST_API->GetOpenOrders();
```

GetClosedOrders

返回已关闭订单的列表。

```
Kraken->REST_API->GetClosedOrders();
```

QueryOrders

查询订单信息。

```
Kraken->REST_API->QueryOrders("1234");
```

GetTradesHistory

返回交易信息数组。

```
Kraken->REST_API->GetTradesHistory();
```

QueryTrades

查询交易信息。

```
Kraken->REST_API->QueryTrades("1234");
```

GetOpenPositions

返回持仓信息。

```
Kraken->REST_API->GetOpenPositions("1234");
```

GetLedgers

返回账本信息的关联数组。

```
Kraken->REST_API->GetLedgers();
```

QueryLedgers

返回账本信息的关联数组。

```
Kraken->REST_API->QueryLedgers("1234");
```

GetTradeVolume

返回交易量信息。

```
Kraken->REST_API->GetTradeVolume();
```

AddExport

添加新的报告导出。

```
Kraken->REST_API->AddExport("Report All Trades");
```

ExportStatus

获取报告状态

```
Kraken->REST_API->ExportStatus();
```

RetrieveExport

根据报告 ID 获取报告。

```
Kraken->REST_API->RetrieveExport("GOCO");
```

RemoveExport

通过报告 id 删除报告。

```
Kraken->REST_API->RemoveExport("GOCO");
```

添加订单

添加新订单

```
pair = asset pair
type = type of order (buy/sell)
ordertype = order type:
    market
    limit (price = limit price)
    stop-loss (price = stop loss price)
    take-profit (price = take profit price)
    stop-loss-profit (price = stop loss price, price2 = take profit price)
    stop-loss-profit-limit (price = stop loss price, price2 = take profit price)
    stop-loss-limit (price = stop loss trigger price, price2 = triggered limit price)
    take-profit-limit (price = take profit trigger price, price2 = triggered limit price)
    trailing-stop (price = trailing stop offset)
    trailing-stop-limit (price = trailing stop offset, price2 = triggered limit offset)
    stop-loss-and-limit (price = stop loss price, price2 = limit price)
    settle-position
price = price (optional. dependent upon ordertype)
price2 = secondary price (optional. dependent upon ordertype)
volume = order volume in lots
leverage = amount of leverage desired (optional. default = none)
oflags = comma delimited list of order flags (optional):
    viqc = volume in quote currency (not available for leveraged orders)
    fcib = prefer fee in base currency
    fciq = prefer fee in quote currency
    nompp = no market price protection
    post = post only order (available when ordertype = limit)
starttm = scheduled start time (optional):
    0 = now (default)
    +n = schedule start time n seconds from now
    n = unix timestamp of start time
expiretm = expiration time (optional):
    0 = no expiration (default)
    +n = expire n seconds from now
    n = unix timestamp of expiration time
userref = user reference id. 32-bit signed number. (optional)
validate = validate inputs only. do not submit order (optional)
optional closing order to add to system when order gets filled:
    close[ordertype] = order type
    close[price] = price
    close[price2] = secondary price
```

```
TsgcHTTPKrakenOrder *oKrakenOrder = new TsgcHTTPKrakenOrder();
oKrakenOrder->Pair = "XBT/USD";
oKrakenOrder->_Type = koshBuy;
oKrakenOrder->OrderType = kothMarket;
oKrakenOrder->Volume = 1;
Kraken->REST_API->AddOrder(oKrakenOrder);
```

CancelOrder

按 ID 取消未结订单

```
Kraken->REST_API->CancelOrder("1234");
```

API Kraken Futures

Kraken Futures

概述

REST API 允许您安全访问 Kraken Futures 账户的方法。REST API 方法示例：

- 请求当前或历史价格信息
- 查看您的账户余额和损益
- 您的保证金参数及预估强制平仓阈值
- 逐个或批量下达或取消订单
- 查看您的未结订单
- 未平仓或交易历史
- 请求提取数字资产

这些方法称为"端点"，在 REST API 部分有详细说明。

Websocket API 允许您安全地建立与 Kraken Futures 平台的通信通道，以实时接收信息。这允许监听更新，而无需持续发送请求。这些频道称为订阅。

部分端点允许执行敏感操作，例如发起数字资产提取。为安全访问这些端点，该 API 使用了由美国国家安全局开发的加密技术。

配置

为了使用 API，您需要生成一对唯一的 **API 密钥**（如果您希望访问私有 API）：

1. 登录您的 **Kraken 期货账户**。
2. 单击右上角的用户名。
3. 从下拉菜单中选择"设置"。
4. 在 API 面板中选择"创建密钥"选项卡。
5. 按下"Create Key"按钮。
6. 查看您的公钥和私钥，并将其记录在安全的地方。

将公钥和私钥复制到组件的 **KrakenOptions** 属性中。

```
KrakenOptions.ApiKey
KrakenOptions.ApiSecret
```

支持的 API

- **WebSockets 公开 API**：连接到公开 WebSocket 服务器。
- **WebSockets 私有 API**：连接到私有 WebSocket 服务器，需要 API Key 和 API Secret 向服务器进行认证。
- **REST 公共 API**：连接到公共 REST 服务器。
- **REST 私有 API**：连接到公共 REST 服务器，需要 API Key 和 API Secret 才能向服务器进行身份验证。

REST API 方法

私有端点

方法	参数	描述

BatchOrder		提交批量订单（下单/取消/修改）。
GetOrderStatus		获取特定订单的状态。
GetPNLCurrencyPreferences		获取 PNL 货币偏好设置。
SetPNLCurrencyPreference		设置符号的盈亏货币偏好。
GetLeverageSettings		获取杠杆偏好设置。
SetLeverageSettings		为某个交易对设置杠杆。

API Kraken Futures | WebSockets 公共 API

连接

URL: `wss://futures.kraken.com/ws/v1`

套接字打开后，您可以通过发送订阅请求消息来订阅公共频道。

方法

行情

此端点返回所有当前上市 Futures 合约和指数的当前市场数据。不需要身份验证。
调用 **SubscribeTicker** 方法订阅行情：

```
SubscribeTicker(ARRAYOFCONST(("PI_XBTUSD")));
```

如果订阅成功，将调用 **OnKrakenFuturesSubscribed** 事件：

```
void OnKrakenFuturesSubscribed(TObject *Sender, string Feed, string ProductId)
{
    DoLog("#subscribed: " + Feed + " " + ProductId);
}
```

通过调用 **UnSubscribeTicker** 方法取消订阅：

```
UnSubscribeTicker(ARRAYOFCONST(("PI_XBTUSD")));
```

如果取消订阅成功，将调用 **OnKrakenFuturesUnSubscribed** 事件：

```
void OnKrakenFuturesUnSubscribed(TObject *Sender, string Feed, string ProductId)
{
    DoLog("#unsubscribed: " + Feed + " " + ProductId);
}
```

如果订阅/取消订阅时出错，将调用 **OnKrakenFuturesError** 事件。

```
void OnKrakenFuturesError(TObject *Sender, string Error)
{
    DoLog("#error: " + Error);
}
```

行情更新将通过 **OnKrakenData** 事件通知。

```
{    "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"
}
```

交易

交易数据流返回已成交交易的信息
调用 **SubscribeTrade** 方法订阅交易数据流。

```
SubscribeTrade(ARRAYOFCONST(("PI_XBTUSD")));
```

如果订阅成功，将调用 **OnKrakenFuturesSubscribed** 事件：

```
void OnKrakenFuturesSubscribed(TObject *Sender, string Feed, string ProductId,)
{
  DoLog("#subscribed: " + Feed + " " + ProductId);
}
```

通过调用 **UnSubscribeTrade** 方法取消订阅：

```
UnSubscribeTrade(ARRAYOFCONST(("PI_XBTUSD")));
```

如果取消订阅成功，将调用 **OnKrakenFuturesUnSubscribed** 事件：

```
void OnkrakenFuturesUnSubscribed(TObject *Sender, string Feed, string ProductId)
{
  DoLog("#unsubscribed: " + Feed + " " + ProductId);
}
```

如果订阅/取消订阅时出错，将调用 **OnKrakenFuturesError** 事件。

```
void OnKrakenFuturesError(TObject *Sender, string Error)
{
  DoLog("#error: " + Error);
}
```

交易更新将在 **OnKrakenData** 事件中通知。

```
{  "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
}
]
```

订单簿

此 feed 返回有关订单簿的信息。
调用 `SubscribeBook` 方法订阅订单簿，须传入 `Symbol`。

```
SubscribeBook(ARRAYOFCONST(("PI_XBTUSD")));
```

如果订阅成功，将调用 **OnKrakenFuturesSubscribed** 事件：

```
void OnKrakenFuturesSubscribed(TObject *Sender, string Feed, string ProductId)
{
  DoLog("#subscribed: " + Feed + " " + ProductId);
}
```

调用 **UnSubscribeBook** 方法取消订阅：

```
UnSubscribeBook(ARRAYOFCONST(("PI_XBTUSD")));
```

如果取消订阅成功，将调用 **OnKrakenFuturesUnSubscribed** 事件：

```
void OnKrakenFuturesUnSubscribed(TObject *Sender, string Feed, string ProductId)
{
    DoLog("#unsubscribed: " + Feed + " " + ProductId);
}
```

如果订阅/取消订阅时出错，将调用 **OnKrakenFuturesError** 事件。

```
void OnKrakenFuturesError(TObject *Sender, string Error)
{
    DoLog("#error: " + Error);
}
```

订单簿更新将通过 **OnKrakenData** 事件通知。

```
{

"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

Ticker 精简订阅源返回已上市产品的 Ticker 信息。
调用 **SubscribeTickerLite** 方法订阅价差数据流。

```
SubscribeTickerLite(ARRAYOFCONST(("PI_XBTUSD")));
```

如果订阅成功，将调用 **OnKrakenFuturesSubscribed** 事件：

```
void OnKrakenFuturesSubscribed(TObject *Sender, string Feed, string ProductId)
{
    DoLog("#subscribed: " + Feed + " " + ProductId);
}
```

调用 **UnSubscribeTickerLite** 方法取消订阅：

```
UnSubscribeTickerLite(ARRAYOFCONST(("PI_XBTUSD")));
```

如果取消订阅成功，将调用 **OnKrakenFuturesUnSubscribed** 事件：

```
void OnKrakenFuturesUnSubscribed(TObject *Sender, string Feed, string ProductId)
{
    DoLog("#unsubscribed: " + Feed + " " + ProductId);
}
```

如果订阅/取消订阅时出错，将调用 **OnKrakenFuturesError** 事件。

```
void OnKrakenFuturesError(TObject *Sender, string Error)
{
    DoLog("#error: " + Error);
}
```

价差更新将在 **OnKrakenData** 事件中通知。

```
{ "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

心跳源按定时间隔发布心跳消息。

```

SubscribeHeartBeat();
UnSubscribeHeartBeat();

```

事件

OnConnect：当 WebSocket 客户端连接到服务器时。

OnKrakenFuturesConnect：在 WebSocket 成功连接后以及服务器发送系统状态时调用。

OnKrakenFuturesSubscribed：成功订阅频道后调用。

OnKrakenFuturesUnSubscribed：成功取消订阅某个频道后调用。

OnKrakenFuturesError：订阅/取消订阅时发生任何错误时调用。

OnKrakenData：每次频道订阅有更新时调用。

API Kraken 期货 | WebSockets 私有 API

连接

URL: `wss://futures.kraken.com/ws/v1`

身份验证

订阅和取消订阅 WebSocket 私有数据流的请求需要使用用户 `api_secret` 签名的挑战消息。

如 WebSocket API Public 部分所示（使用 `api_key`），可获取到挑战值。

经身份验证的请求必须以 JSON 格式同时包含原始挑战消息（`original_challenge`）和已签名的挑战（`signed_challenge`）。

为了获取 WebSocket Challenge，必须在 Kraken Options 组件中设置 API Key 和 API Secret，即 Kraken 在您账户中提供的 API 密钥

```
Kraken->ApiKey = "api key";
Kraken->ApiSecret = "api secret";
```

方法

详细的未成交订单

此订阅频道发布用户未结订单的相关信息，并额外提供所有因未能与订单簿撮合而失败的只挂单（post-only）订单的详细信息。

```
SubscribeOpenOrdersVerbose();
```

之后，您可以取消订阅 `OpenOrdersVerbose`，调用 **`UnSubscribeOpenOrdersVerbose`** 方法

```
UnSubscribeOpenOrdersVerbose();
```

服务器的响应示例

```
{
  'feed': 'open_orders_verbose_snapshot',
  {
    'instrument': 'PI_XBTUSD',
    'account': '0f9c23b8-63e2-40e4-9592-6d5aa57c12',
    'time': 1567428848005,
    'last_update': ...
  }
}
```

未平仓头寸

此订阅源发布用户账户的未平仓头寸。

```
SubscribeOpenPositions();
```

稍后，您可以通过调用 **`UnSubscribeOpenPositions`** 方法取消订阅 `OpenPositions`

```
UnSubscribeOpenPositions();
```

服务器的响应示例

```
{
  "feed": "open_positions",
  "account": "DemoUser",
  "positions": [{
    "instrument": "fi_xbtusd_180316",
    "balance": 2000.0,
    "entry_price": 11675.86541981,
    "mark_price": 11090.0,
    "index_price": 12290.5500000000001,
    "pnl": -0.00905299
  }]
}
```

账户日志

此订阅数据流发布账户信息。

```
SubscribeAccountLog();
```

之后，您可以调用 **UnSubscribeAccountLog** 方法取消订阅 AccountLog

```
UnSubscribeAccountLog();
```

服务器的响应示例

```
{
  '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}]
}
```

成交明细

此订阅源发布成交信息。

```
SubscribeFills();
```

之后，您可以通过调用 **UnSubscribeFills** 方法取消订阅 Fills

```
UnSubscribeFills();
```

服务器的响应示例

```
{
  "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"
    }
  ]
}
```

未结订单

此订阅推送用户未结订单的相关信息。

```
SubscribeOpenOrders();
```

之后，您可以通过调用 **UnSubscribeOpenOrders** 方法取消订阅 OpenOrders

```
UnSubscribeOpenOrders();
```

服务器的响应示例

```
{
  "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"
}
]
}

```

账户余额与保证金

此订阅数据流返回客户账户的余额和保证金信息。

```
SubscribeAccountBalanceAndMargins();
```

稍后，您可以调用 **UnSubscribeAccountBalanceAndMargins** 方法取消订阅账户余额

```
UnSubscribeAccountBalanceAndMargins();
```

服务器的响应示例

```

{

"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

}

```

通知

此订阅源向客户端发布通知。

```
SubscribeNotifications();
```

稍后，您可以调用 **UnSubscribeNotifications** 方法取消订阅通知

```
UnSubscribeNotifications();
```

服务器的响应示例

```

{

"feed": "notifications_auth",

"notifications": [

```

```
{  
  "id":5,  
  "type":"market",  
  "priority":"low",  
  "note":"描述通知的说明。",  
  "effective_time":1520288300000  
},  
  ...  
]  
}
```

API Kraken Futures | REST 公共 API

连接

URL: <https://futures.kraken.com/derivatives/api/v3>

Kraken Futures 公共 API 不需要任何身份验证。

配置

唯一的配置是是否为 REST HTTP 请求启用日志。如果您想将所有 HTTP 请求/响应保存到文本文件日志中，请启用 HTTPLogOptions。

事件

OnKrakenHTTPException: 当通过 REST API 执行 HTTP 请求时出现任何异常，将调用此事件。

方法

GetFeeSchedules

此端点列出所有费用计划，无需身份验证。

```
KrakenFutures->REST_API->GetFeeSchedules();
```

订单簿

此端点返回当前上市期货合约的完整非累积订单簿。

```
KrakenFutures->REST_API->GetOrderBook("PI_XBTUSD");
```

行情

此端点返回所有当前上市 Futures 合约和指数的当前市场数据。

```
KrakenFutures->REST_API->GetTickers();
```

仪器

该端点返回所有当前上市期货合约和指数的规格说明。

```
KrakenFutures->REST_API->GetInstruments();
```

历史

此端点返回从指定 lastTime 值起的最后 100 笔交易记录——如果未指定值，则返回最后 100 笔交易。此端点仅返回最多 7 天内的交易历史，从调用时算起，或从最后一次交易引擎发布以来（以较早者为准）。

```
KrakenFutures->REST_API->GetHistory("PI_XBTUSD");
```

API Kraken Futures | REST 私有 API

连接

URL: <https://futures.kraken.com/derivatives/api/v3>

身份验证

REST 私有 API 需要 API Key 和 API Secret，这些值由 Kraken 在您的账户中提供。

```
Kraken->ApiKey = "api key";  
Kraken->ApiSecret = "api secret";
```

方法

EditOrderByOrderId

此端点允许编辑当前上市的期货合约的现有订单。

aOrderId：您希望编辑的订单 ID

aSize：与订单关联的数量

aLimitPrice：与订单关联的限价。

aStopPrice：与止损订单关联的止损价格。如果旧订单类型为止损，则为必填。

```
KrakenFutures->REST_API->EditOrderByOrderId("Order_Id", 2, 1000);
```

EditOrderByCliOrderId

此端点允许编辑当前上市的期货合约的现有订单。

aCliOrderId：用户指定的订单标识符，必须全局唯一。

aSize：与订单关联的数量

aLimitPrice：与订单关联的限价。

aStopPrice：与止损订单关联的止损价格。当订单类型为 Stop 时为必填项。

```
KrakenFutures->REST_API->EditOrderByCliOrderId("Cli_Order_Id", 2, 1000);
```

SendMarketOrder

此端点允许您发送市价单。

aSide：订单方向：买入或卖出。

aSymbol：期货的交易对

aSize：与订单关联的数量。

```
KrakenFutures->REST_API->SendMarketOrder(kosfBuy, "PI_XBTUSD", 1);
```

SendLimitOrder

此端点允许您发送限价单。

aSide：订单方向：买入或卖出。
aSymbol：期货的交易对
aSize：与订单关联的数量。
aLimitPrice：与订单关联的限价。

```
KrakenFutures->REST_API->SendLimitOrder(kosfBuy, "PI_XBTUSD", 1, 1000);
```

SendStopOrder

此端点允许您发送止损订单。

aSide：订单方向：买入或卖出。
aSymbol：期货的交易对
aSize：与订单关联的数量。
aStopPrice：与止损订单关联的止损价格。
aLimitPrice：与订单关联的限价。

```
KrakenFutures->REST_API->SendStopOrder(kosfBuy, "PI_XBTUSD", 1, 1000, 900);
```

SendTakeProfitOrder

此端点允许您下达止盈订单。

aSide：订单方向：买入或卖出。
aSymbol：期货的交易对
aSize：与订单关联的数量。
aStopPrice：与止损订单关联的止损价格。
aLimitPrice：与订单关联的限价。

```
KrakenFutures->REST_API->SendTakeProfitOrder(kosfBuy, "PI_XBTUSD", 1, 1000, 900);
```

SendOrder

此端点允许为当前上市的期货合约发送限价、止损、止盈或立即或取消订单。

OrderType：请选择以下之一：kotfLMT、kotfPOST、kotfMKT、kotfSTP、kotfTAKE_PROFIT、kotfIOC。
Symbol：期货的交易对符号
Side：订单方向（买入或卖出）。
Size：与订单关联的数量。
StopPrice：与止损订单关联的止损价格。
LimitPrice：与订单关联的限价。
TriggerSignal：若下单止损或止盈订单，则为触发所用信号，从以下选项中选择：kotsMark、kotsIndex、kotsLast
CliOrderId：由用户指定的订单标识，必须全局唯一。
ReduceOnly：若希望订单仅减少现有仓位，则设为 true。任何会增加现有仓位的订单都将被拒绝。默认为 false。

```
TsgcHTTPKrakenFuturesOrder *oOrder = new TsgcHTTPKrakenFuturesOrder(this);
try
{
    oOrder->Side = kosfBuy;
    oOrder->Symbol = "PI_XBTUSD";
    oOrder->OrderType = kotfMKT;
    oOrder->Size = 1;
    KrakenFutures->REST_API->SendOrder(oOrder);
}
__finally
{
```

```
oOrder->Free;
}
```

CancelOrderByOrderId

此端点允许取消期货合约的未结订单。

aOrderId：您希望编辑的订单 ID

```
KrakenFutures->REST_API->CancelOrderByOrderId("Order_Id");
```

CancelOrderByCliOrderId

此端点允许取消期货合约的未结订单。

aCliOrderId：用户指定的订单标识符，必须全局唯一。

```
KrakenFutures->REST_API->CancelOrderByCliOrderId("Cli_Order_Id");
```

GetFills

此端点返回所有期货合约的已成交订单信息。

aLastFillDate：如果未提供，返回任意期货合约的最后 100 条成交记录。如果提供，返回 lastFillTime 之前的 100 条记录。

```
KrakenFutures->REST_API->GetFills("2020-07-22T13:45:00.000Z");
```

传输

此端点允许您在具有相同抵押货币的两个保证金账户之间，或在保证金账户与现金账户之间转移资金。

aFromAccount：要移出资金的现金或保证金账户名称。

aToAccount：要将资金转入的现金或保证金账户名称。

aUnit：要转移的单位。

aAmount：要转账的金额。

```
KrakenFutures->REST_API->Transfer("FI_XBTUSD", "cash", "xbt", 1.5);
```

GetOpenPositions

此端点返回期货合约中所有未平仓头寸的规模和平均入场价格，包括已到期但尚未结算的期货合约。

```
KrakenFutures->REST_API->GetOpenPositions();
```

GetNotifications

该端点提供平台的通知。

```
KrakenFutures->REST_API->GetNotifications();
```


GetAccounts

此端点返回与您所有 Kraken Futures 账户相关的关键信息，这些账户可能是现金账户或保证金账户。包括数字资产余额、合约余额、保证金要求、保证金触发估算，以及可用资金、未结仓位盈亏和投资组合价值等辅助信息。

```
KrakenFutures->REST_API->GetAccounts();
```

CancelAllOrders

此端点允许取消期货合约的未结订单。

Symbol：要取消所有未成交订单的期货产品（可选）

```
KrakenFutures->REST_API->CancelAllOrders();
```

CancelAllOrdersAfter

此端点提供了一种"死人开关"机制，以保护客户端免受网络故障的影响。客户端可以发送一个带有超时秒数的请求，这将触发倒计时计时器，在超时到期时取消所有客户端订单。

aTimeout：以秒为单位指定的超时时间。

```
KrakenFutures->REST_API->CancelAllOrdersAfter(60);
```

GetOpenOrders

此端点返回所有期货合约的所有挂单信息。

```
KrakenFutures->REST_API->OpenOrders();
```

GetHistoricalOrders

此端点返回账户的历史订单。

aSince：起始日期时间

aBefore：截止 DateTime

aSort："asc" 表示升序，"desc" 表示降序

aContinuationToken：来自先前响应的延续令牌，可用于调用以返回下一组可用结果

```
KrakenFutures->REST_API->GetHistoricalOrders(Now, Now - 5);
```

GetHistoricalTriggers

此端点允许返回账户上的历史触发器。

aSince：起始日期时间

aBefore：截止 DateTime

aSort："asc" 表示升序，"desc" 表示降序

aContinuationToken：来自先前响应的延续令牌，可用于调用以返回下一组可用结果

```
KrakenFutures->REST_API->GetHistoricalTriggers(Now, Now - 5);
```

GetHistoricalExecutions

此端点允许获取账户上的历史成交记录。

aSince：起始日期时间

aBefore: 截止 DateTime

aSort："asc" 表示升序，"desc" 表示降序

aContinuationToken：来自先前响应的延续令牌，可用于调用以返回下一组可用结果

```
KrakenFutures->REST_API->GetHistoricalExecutions(Now, Now - 5);
```

WithdrawalToSpotWallet

此端点允许提交请求，将数字资产从 Kraken Futures 钱包提款到您的 Kraken Spot 钱包。

aCurrency：要提款的数字资产，例如 xbt 或 xrp。

aAmount：要提取的货币金额。

```
KrakenFutures->REST_API->WithdrawalToSpotWallet("xbt", 1000);
```

GetFeeScheduleVolumes

此端点返回您 30 天的 USD 交易量。

```
KrakenFutures->REST_API->GetFeeScheduleVolumes();
```

GetAccountLogCSV

此端点允许客户端下载其账户日志的 CSV 文件。

```
KrakenFutures->REST_API->GetAccountLogCSV();
```

API Pusher

Pusher

Pusher 是一个基于 WebSocket 协议的简单可靠平台，提供灵活的发布/订阅消息、在线用户列表（Presence）和身份验证等功能。

Pusher WebSocket API 版本为 7。

数据通过 WebSocket 以包含 UTF-8 编码 JSON 的文本数据双向传输（不支持二进制 WebSocket 帧）。

您可以调用 **Ping** 方法来测试与服务器的连接。基本上，收到对方的任何消息都表示连接处于活跃状态。在没有任何消息的情况下，任何一方都可以通过发送 ping 消息来检查另一方是否响应，另一方应以 pong 回应。

连接之前，您必须填写以下字段：

```
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)
```

重要

Pusher 要求 WebSocket 客户端使用前面的字段（key、cluster 等）构建 URL 进行连接，这些字段在将客户端分配给 Pusher 组件时用于构建 URL。因此，为确保 URL 构建正确，请在填写完 Pusher 配置字段后再设置客户端。以下是伪代码示例：

```
// configure pusher fields
pusher.cluster = ...
pusher.key = ...
// 设置客户端
pusher.client = websocket client
// start connection
websocket client.Active = true;
```

成功连接后，将触发 **OnPusherConnect** 事件，并获得以下字段：

- Socket ID：已连接客户端的唯一标识符。
- Timeout：服务器不活跃多少秒后，客户端应发起 ping 消息（由组件自动处理）。

发生错误时，将触发 **OnPusherError**，并提供有关错误的信息。错误可能由 Pusher 响应无效身份验证、无效命令等情况发送。

4000-4099

表示由于 Pusher 关闭连接而发生的错误，使用相同参数尝试重新连接将不会成功。

- 4000: 应用程序仅接受 SSL 连接，请使用 wss:// 重新连接
- 4001: 应用程序不存在
- 4003: 应用程序已禁用
- 4004: 应用程序超出连接配额
- 4005: 路径未找到
- 4006: 版本字符串格式无效
- 4007: 不支持的协议版本
- 4008: 未提供协议版本

4100-4199

表示由于错误导致 Pusher 关闭连接，客户端可在 1 秒或更长时间后重新连接。

- 4100: 超出容量

4200-4299

表示因连接被 Pusher 关闭而导致的错误，客户端可以立即重新连接。

- 4200: 立即通用重连
- 4201: 未收到 Pong 回复：已向客户端发送 ping，但未收到回复。请参阅 ping 和 pong 消息。
- 4202: 因不活动而关闭：客户端长时间处于非活动状态（目前为 24 小时），且客户端不支持 ping。请升级到较新的 WebSocket 草案或实现第 5 版及以上版本的此协议。

4300-4399

任何其他类型的错误。

- 4301: 由于速率限制，客户端事件被拒绝

通道

频道是 Pusher 中的基本概念。每个应用程序有多个频道，每个客户端可以选择订阅哪些频道。

频道提供：

- 一种过滤数据的方式。例如，在聊天应用程序中，可能有一个供想要讨论"狗"的人使用的频道
- 一种控制不同信息流访问权限的方式。例如，项目管理应用程序可授权用户获取"projectX"的更新。

强烈建议使用频道过滤数据，而不是通过事件实现。这是因为发布到频道的所有事件都会发送给所有订阅者，无论其绑定的事件类型如何。

频道无需显式创建，按需在客户端请求时实例化。这意味着创建频道非常简单，只需告诉客户端订阅即可。

支持以下类型的频道：

- **公共频道**可由任何知道其名称的人订阅
- **私有频道**引入了一种机制，让您的服务器能够控制对您广播数据的访问。
- **Presence 频道**是私有频道的扩展。它们允许您在订阅时注册用户信息，并让频道中的其他成员知道谁在线。
- **缓存频道**会记住最近一次触发的事件，并将其作为第一个事件发送给新订阅者（支持公共、私有和 Presence 变体）
- **Private-Encrypted 频道**使用 NaCl secretbox 提供端到端加密，确保即使 Pusher 也无法读取消息数据。

公共频道

公共频道应用于可公开访问的数据，因为它们不需要任何形式的授权即可订阅。

您可以随时订阅和取消订阅频道，无需等待 Pusher 完成连接。

示例：订阅频道"my-channel"。

```
CBuilder
APIPusher->Subscribe("my-channel");
```

如果订阅成功，**OnPusherSubscribe** 事件将触发；若发生错误，您将在 **OnPusherError** 事件中收到消息。

来自自己订阅频道的所有消息都将通过 **OnPusherEvent** 事件接收。

当调用 Publish 方法且通道为公共通道时，组件不使用 WebSocket 协议，而是使用 HTTP 协议并调用 TriggerEvent 方法（WebSocket 协议不允许发布消息）。

私有频道

需要 Indy 10.5.7 或更高版本

当需要以某种方式限制对频道的访问时，应使用私有频道。用户必须获得授权才能订阅私有频道。

示例：订阅频道 "my-private-channel"。

```
CBuilder
APIPusher->Subscribe("my-private-channel", pscPrivateChannel);
```

如果订阅成功，**OnPusherSubscribe** 事件将触发；若发生错误，您将在 **OnPusherError** 事件中收到消息。

来自自己订阅频道的所有消息都将通过 **OnPusherEvent** 事件接收。

Presence 频道

需要 Indy 10.5.7 或更高版本

Presence 频道建立在私有频道的安全性之上，并公开了一个额外功能：感知谁订阅了该频道。这使得在您的应用程序中构建聊天室和"谁在线"类型的功能变得非常容易。想象一下聊天室、文档上的协作者、浏览同一网页的人、游戏中的竞争者等场景。

Presence 频道的订阅方式与私有频道相同，但频道名称必须以 presence- 为前缀。与私有频道一样，系统会向可配置的认证 URL 发起 HTTP 请求，以确定当前用户是否有权限访问该频道。

用户订阅和取消订阅频道的相关信息，随后可通过绑定到在线频道上的事件来访问，当前订阅该频道的用户状态可通过 channel.members 属性获取。

示例：订阅频道 "my-presence-channel"。

```
APIPusher->Subscribe("my-presence-channel", pscPresenceChannel,
    '{"user_id":"","John_Smith","user_info":{"name":"","John Smith"}}');
```

如果订阅成功，**OnPusherSubscribe** 事件将触发；若发生错误，您将在 **OnPusherError** 事件中收到消息。

来自自己订阅频道的所有消息都将通过 **OnPusherEvent** 事件接收。

缓存频道

缓存频道会记住上次触发的事件，并将其作为第一个事件发送给新订阅者。

当在缓存频道上触发事件时，Pusher Channels 会缓存此事件。当客户端订阅缓存频道时，如果存在缓存值，则将其作为该频道的第一个事件发送给客户端。此行为帮助开发者无需添加额外逻辑即可提供初始状态。

支持以下缓存频道：

- 公共缓存频道
- 私有缓存频道
- Presence 缓存频道

示例：订阅公共缓存频道"my-cache-channel"。

```
APIPusher->Subscribe("my-cache-channel", pscCacheChannel);
```

如果订阅成功，**OnPusherSubscribe** 事件将触发；若发生错误，您将在 **OnPusherError** 事件中收到消息。

来自已订阅频道的所有消息都将通过 **OnPusherEvent** 事件接收。

如果订阅缓存频道时没有缓存事件，将引发 **OnPusherCacheMiss** 事件，提供频道名称。这允许您的应用程序处理无缓存数据可用的情况。

私有加密频道

私有加密通道为消息提供端到端加密。与私有通道一样，它们需要身份验证，但所有数据负载均使用 NaCl secretbox 加密，因此只有授权订阅者才能读取内容。即使是 Pusher 本身也无法解密这些消息。

要使用私有加密频道，您必须在身份验证期间提供 **SharedSecret**。共享密钥用于加密和解密消息数据。

示例：订阅私有加密频道 "my-encrypted-channel"。

```
APIPusher->Subscribe("my-encrypted-channel", pscPrivateEncryptedChannel);
```

还提供私有加密缓存变体，将加密与缓存通道行为相结合：

```
APIPusher->Subscribe("my-encrypted-cache-channel", pscPrivateEncryptedCacheChannel);
```

在使用带有私有加密频道的 **OnPusherAuthentication** 事件时，您可以在响应对象上设置 **SharedSecret** 属性以提供加密密钥：

```
void OnPusherAuthenticationEvent(TObject *Sender,
    TsgcWSPusherRequestAuthentication *AuthRequest,
    TsgcWSPusherResponseAuthentication *AuthResponse)
{
    AuthResponse->SharedSecret = "your-shared-secret-key";
}
```

在线状态事件

Presence 频道提供额外的事件，当用户加入或离开频道时通知您的应用程序，并允许您跟踪订阅数量。

OnPusherMemberAdded

当新成员订阅 Presence 频道时触发。提供频道名称、用户 ID 以及加入成员的用户信息。

```
void PUSHERPusherMemberAdded(TObject *Sender,
    String Channel, String UserId, String UserInfo)
{
    Log("Member joined: " + UserId + " on " + Channel);
}
```

OnPusherMemberRemoved

当某成员取消订阅 presence 频道时触发。提供频道名称、用户 ID 以及已离开成员的用户信息。

```
void PUSHERPusherMemberRemoved(TObject *Sender,
    String Channel, String UserId, String UserInfo)
{
    Log("Member left: " + UserId + " on " + Channel);
}
```

OnPusherSubscriptionCount

当频道上的订阅数量发生变化时触发。提供频道名称和当前订阅者数量。此事件必须在您的 Pusher 仪表板上启用。

```
void PUSHERPusherSubscriptionCount(TObject *Sender,
    String Channel, int SubscriptionCount)
{
    Log(Channel + " has " + IntToStr(SubscriptionCount) + " subscribers");
}
```

OnPusherCacheMiss

当订阅一个没有缓存事件的缓存频道时触发。提供频道名称。这允许您的应用程序处理没有缓存数据可用的情况，例如从另一个来源获取数据。

```
void PUSHERPusherCacheMiss(TObject *Sender, String Channel)
{
    Log("Cache miss on: " + Channel);
}
```

发布消息

您不仅可以从订阅的频道接收消息，还可以向其他已订阅的用户发送消息。

调用 **Publish** 方法，向频道的所有订阅用户发送消息。

示例：向 "my-channel" 的所有订阅用户发送事件。

```
APIPusher->Publish("my-event", "my-channel");
```

每个客户端（连接）每秒最多发布 10 条消息。超过此速率限制的任何事件将被 Pusher API 拒绝。这不是系统问题，而是客户端问题。如果一个通道中的 100 个客户端都以此速率发送消息，那么每个客户端每秒也需要处理 1,000 条消息！尽管某些现代浏览器可能能够处理这种情况，但这很可能不是一个好主意。

REST API

API 托管在 <http://api-CLUSTER.pusher.com>，其中 CLUSTER 替换为您自己应用的集群（例如 eu）。

HTTP 状态码用于指示请求的成功或其他状态。以下状态码是常见的：

200 请求成功。正文将包含响应数据的 JSON 哈希值

400 错误：详情见响应体

401 身份验证错误：响应体将包含说明。

403 Forbidden：应用程序已禁用或超出消息配额

已实现以下 REST API 函数。

函数	描述
TriggerEvent	在指定频道上触发新事件。支持可选的 SocketId（用于排除某个客户端）和 Info 参数。
TriggerBatchEvents	在单个 HTTP 请求中触发多个事件。接受事件对象的 JSON 数组。
GetChannels	提供所有活跃频道的列表，支持可选的 FilterByPrefix 和 Info 参数。

GetChannel	提供有关特定通道的信息。支持可选的 Info 参数。
GetUsers	提供连接到某个频道的所有用户列表。
TerminateUserConnections	终止给定用户 ID 的所有连接。

TriggerEvent

在一个或多个频道上触发事件。需要提供事件名称、频道名称和数据有效负载。

参数	描述
aEventName	要触发的事件名称。
aChannel	用于触发事件的频道名称。
aData	事件数据（JSON 字符串）。
aSocketId（可选）	要从接收事件中排除的套接字 ID。可用于防止发送者接收到自己的消息。
aInfo（可选）	响应中要包含的属性逗号分隔列表（例如 "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

在单次 API 调用中触发多个事件，比分别为每个事件发送请求更为高效。batch 参数必须是包含事件对象数组的 JSON 字符串，每个对象均包含"channel"、"name"和"data"字段。

```
APIPusher->TriggerBatchEvents(
    "[{"channel\":\"my-channel\",\"name\":\"my-event\",\"data\":\"hello\"}, {"channel\":\"my-channel-2\",\"name\":\"my-event\",\"data\":\"world\"}]");
```

GetChannels

返回活动频道列表。支持可选参数以过滤结果并请求额外信息。

参数	描述
aFilterByPrefix（可选）	按名称前缀筛选频道（例如，"presence-" 以仅列出 presence 频道）。
aInfo（可选）	响应中包含的属性的逗号分隔列表（例如"user_count"）。

```
// get all channels
APIPusher->GetChannels();

// get only presence channels with user count
APIPusher->GetChannels("presence-", "user_count");
```

GetChannel

返回有关特定频道的信息。

参数	描述
aChannel	要获取相关信息的频道名称。
alInfo（可选）	以逗号分隔的要包含的属性列表（例如 "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

返回连接到 Presence 频道的用户列表。频道名称必须包含完整前缀（例如 "presence-my-channel"）。

```
APIPusher->GetUsers("presence-my-channel");
```

TerminateUserConnections

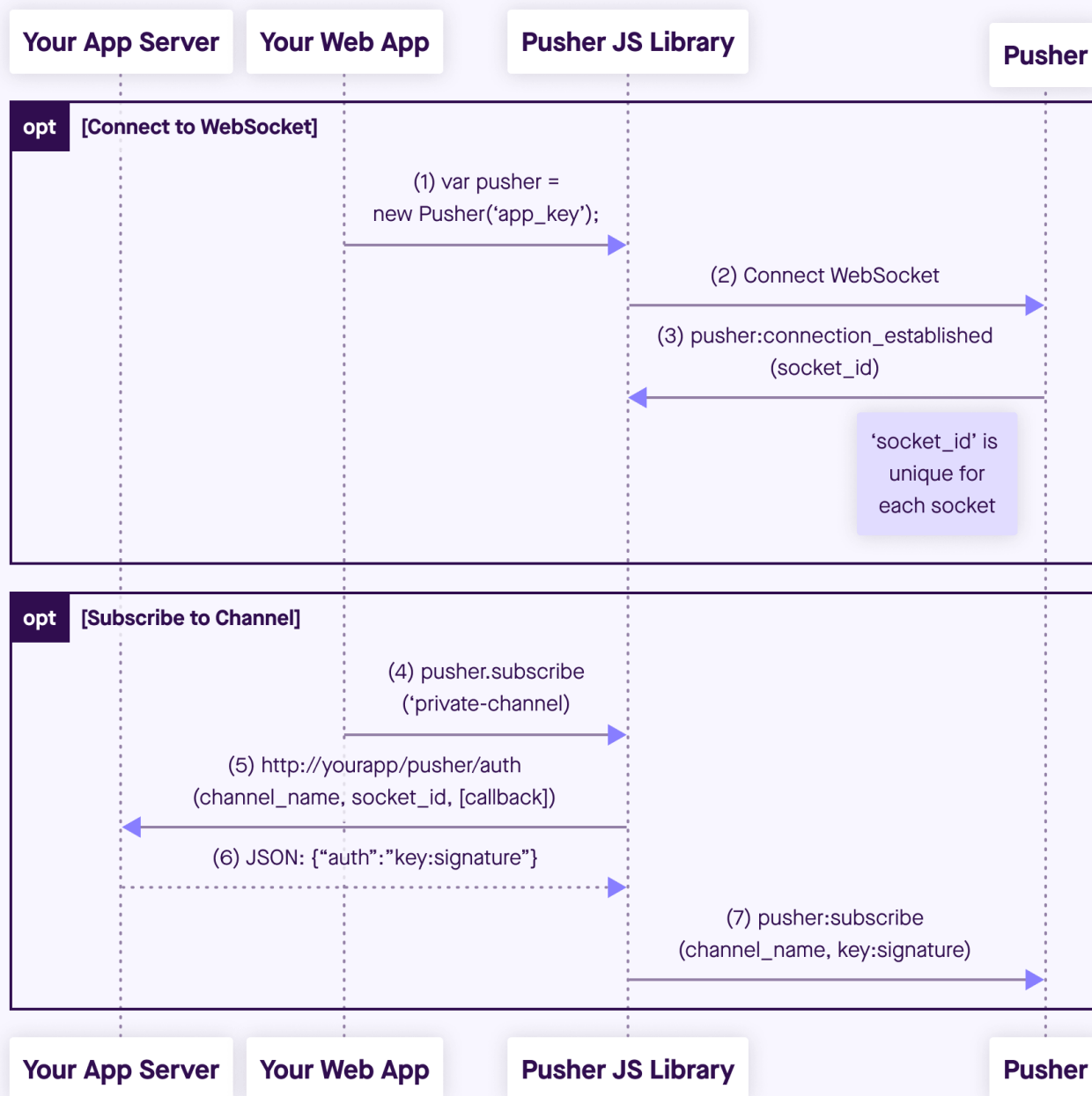
终止指定用户建立的所有连接。可用于强制特定用户从所有通道断开连接。用户 ID 必须与用户订阅 Presence 通道时使用的"user_id"匹配。

```
APIPusher->TerminateUserConnections("1234");
```

自定义身份验证

Pusher 仅允许订阅私有或存在频道；如果连接提供了身份验证令牌，则可限制访问权限。

您可以使用 **OnPusherAuthentication** 事件构建自己的认证流程，此事件在订阅消息使用 Pusher 提供的密钥签名之前调用。此事件有 2 个参数，一个包含 SocketId、频道名等字段的认证请求，您的认证服务器可以使用这些字段来决定是否认证请求。以下截图显示了 Pusher 认证流程



当客户端连接到 Pusher 服务器时，将发送 Pusher 提供的 Key，服务器返回一个标识 ID（socket_id）。当客户端订阅私有（或 presence）频道时，sgcWebSockets 客户端使用 Pusher 提供的 Secret Key 创建包含在订阅消息中的签名。通过 OnPusherAuthentication 事件，您可以捕获签名消息所需的字段，实现您自己的身份验证方法，如果成功，则返回签名，该签名将包含在订阅消息中并发送到服务器。

示例：

```

TsgcWebSocketClient *oClient = new TsgcWebSocketClient(NULL);
TsgcWSAPI_Pusher *oPusher = new TsgcWSAPI_Pusher(NULL);
oPusher->Client = oClient;
oPusher->Cluster = "eu";
Pusher->Name = "js";
Pusher->Version = "4.1";
Pusher->TLS = true;
Pusher->Key = "9c3b7ef25qe97a00116c";
Pusher->Secret = ""; // the secret key is not known by the client, only by the authentication module

oPusher->OnPusherAuthentication = OnPusherAuthenticationEvent;

private void OnPusherAuthenticationEvent(TObject *Sender, TsgcWSPusherRequestAuthentication
*AuthRequest, TsgcWSPusherResponseAuthentication *AuthResponse)
{
    // if the authentication request is successful return the signature
    if (CustomAuthentication(AuthRequest->Channel, AuthRequest->SocketID))
    {
        AuthResponse->Signature = GetCustomAuthenticationSignature();
    }
}
  
```

```
}  
}
```

签名的格式为：
私有频道：key:HMAC256(SocketID, ChannelName)
Presence 频道：密钥：HMAC256(SocketID, ChannelName, Data)
TsgcWSPusherResponseAuthentication 对象提供以下属性：

属性	描述
密钥	用于计算 HMAC 签名的 Pusher 密钥。若已配置 Pusher.Secret，则预先填充该值。
签名	计算出的身份验证签名。如果留空，组件将使用 Secret 自动计算。
SharedSecret	私有加密频道的共享密钥。订阅 pscPrivateEncryptedChannel 或 pscPrivateEncryptedCacheChannel 时需要。用于消息数据的端到端加密。

API Bitmex

Bitmex

Bitmex 是一个加密货币交易所和衍生品交易平台。

以下 API 受支持：

1. **WebSocket 流**：允许您订阅某些方法并实时获取数据。服务器将事件推送给订阅者。使用 WebSocket 作为协议。
2. **REST API**：客户端可向服务器请求市场和账户数据。需要 API Key 和 Secret 进行身份验证，并使用 HTTPS 协议。

属性

Bitmex API 有两种方法：公开和私有。公开方法无需身份验证即可访问，例如获取行情价格。部分与用户数据相关的方法是私有的，需要使用 Bitmex API 密钥。

- **ApiKey**：您可以在 Bitmex 账户中申请新的 API 密钥，将值复制到此属性即可。
- **ApiSecret**：API 的密钥，请妥善保管。
- **TestNet**：如果启用，将连接到 Bitmex 演示账户（默认为 false）。
- **HTTPLogOptions**：将 HTTP 请求日志存储到文本文件中
 - **Enabled**：若启用，将存储 WebSocket API 的所有 HTTP 请求。
 - **FileName**：存储日志的文件完整路径

最常见的用途

- **WebSockets API**
 - [如何连接到 WebSocket API](#)
 - [如何订阅 WebSocket 频道](#)
- **REST API**
 - [如何下 Bitmex 订单](#)

WebSocket API

订阅/取消订阅

BitMEX 允许订阅实时数据。此访问一旦连接就不受速率限制，是将最新数据传递给程序的最佳方式。在某些主题中，您可以传入 Symbol 以按交易对过滤事件，例如：trades、quotes.....

以下订阅主题无需身份验证即可使用：

- **btmAnnouncement**：站点公告
- **btmChat**：讨论区聊天
- **btmConnected**：已连接用户/机器人的统计信息
- **btmFunding**：掉期资金费率更新。每个资金结算间隔（通常为 8 小时）发送一次
- **btmInstrument**：包含成交量及买/卖价的品种更新
- **btmInsurance**：每日保险基金更新。
- **btmLiquidation**：进入订单簿时的强平订单
- **btmOrderBookL2_25**：Level 2 订单簿前 25 档
- **btmOrderBookL2**：完整的 2 级订单簿
- **btmOrderBook10**：使用传统全量推送的前 10 档订单簿
- **btmPublicNotifications**：系统范围通知（用于短期消息）
- **btmQuote**：订单簿顶层

- **btmQuoteBin1m**：1 分钟报价分档
- **btmQuoteBin5m**：5 分钟报价区间
- **btmQuoteBin1h**：1 小时报价桶
- **btmQuoteBin1d**：1 天报价仓
- **btmSettlement**：结算
- **btmTrade**：实时交易
- **btmTradeBin1m**：1 分钟交易仓
- **btmTradeBin5m**：5 分钟交易分档
- **btmTradeBin1h**：1 小时交易 bin
- **btmTradeBin1d**：1 天交易箱

以下主题需要身份验证：

- **btmAffiliate**：附属状态，例如总推荐用户数和返佣比例
- **btmExecution**：单次成交；一个订单可有多次成交
- **btmOrder**：订单实时更新
- **btmMargin**：您当前账户余额和保证金要求的更新
- **btmPosition**：您的仓位更新
- **btmPrivateNotifications**：个人通知——目前未使用
- **btmTransact**：充值/提现更新
- **btmWallet**：比特币地址余额数据，包括总存款和提款

收到消息示例：

```
{
  "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"},
  ]
}
```

身份验证

如果您希望订阅用户锁定流，必须先进行身份验证。请注意，无效的身份验证将关闭连接。

BitMEX API 的使用需要 API Key。

永久 API 密钥可以锁定到 IP 地址范围，并可随时撤销，而不会危及您的主要凭据。它们也不需要续期。

要使用 API Key 认证，您必须在账户中生成一个 API Key。

在订阅任何已认证主题之前，请先调用方法 **Authenticate**。

CancelAllAfter (死人开关)

CancelAllAfter 方法实现了"死亡开关" (Dead Man's Switch) 功能。调用时传入超时值 (毫秒)，它将指示服务器在超时时间内未收到后续 **CancelAllAfter** 调用时取消所有未结订单。这在网络断连时确保订单被取消非常有用。

REST API

方法	描述
GetExecutions	这返回所有原始交易，包括订单开立和取消以及订单状态更改。
GetExecutionsTradeHistory	这将返回更有针对性的交易记录。
GetInstruments	此接口返回所有工具和指数，包括已结算或未上市的。如果您要查询单个工具或使用复杂过滤条件，请使用此端点。
GetOrders	仅获取未结订单
PlaceOrder	使用 TsgcHTTPBitmexOrder 对象下原始订单。
PlaceMarketOrder	下一个新市价单。
PlaceLimitOrder	下一个 LIMIT 限价单。
PlaceStopOrder	下达新的 STOP 订单。
PlaceStopLimitOrder	下一个新的 STOPLIMIT 订单。
AmendOrder	修改现有订单。
CancelOrder	取消活跃订单。
CancelAllOrders	取消所有活跃订单。
CancelAllOrdersAfter	一段时间后取消所有订单。
ClosePosition	关闭一个未平仓位。
GetOrderBook	以垂直格式获取当前订单簿
GetPosition	获取您的持仓。
SetPositionIsolate	启用隔离保证金或全仓保证金 (按仓位)。
SetPositionLeverage	按持仓选择杠杆。
SetPositionRiskLimit	更新您的风险限额。
SetPositionTransferMargin	将权益转入或转出某个头寸。
GetQuotes	获取报价
GetTrades	获取交易
GetFunding	获取资金数据。
GetInsurance	获取保险基金数据。
GetTradeBucketed	获取可配置 bin 大小的分桶交易数据 (OHLCV)。
GetQuoteBucketed	获取可配置桶大小的分段报价数据。
GetSettlement	获取结算数据。
GetLiquidation	获取强制平仓订单。
GetInstrumentIndices	获取仪器指数。
GetInstrumentCompositeIndex	获取金融工具的综合指数数据。
GetStats	获取全市场统计数据。
GetStatsHistory	获取历史交易所统计数据。
GetStatsHistoryUSD	获取历史美元汇率统计数据。
GetUserMargin	获取您的账户保证金数据。
GetUserWallet	获取您的钱包信息。

GetUserWalletHistory	获取您的钱包交易历史。
GetUserWalletSummary	获取钱包摘要。

Bitmex | 连接 WebSocket API

要连接到 Bitmex WebSocket API，只需创建一个新的 Bitmex API 客户端并附加到 TsgcWebSocketClient 即可。请参见下方示例：

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bitmex oBitmex = new TsgcWSAPI_Bitmex();
oBitmex->Client = oClient;
oClient->Active = true;
```

Bitmex | 订阅 WebSocket 频道

Bitmex 提供多种频道供订阅，以获取市场数据、订单等的实时更新。以下是订阅 Trade 频道的示例：

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bitmex oBitmex = new TsgcWSAPI_Bitmex();
oBitmex->Client = oClient;
oBitmex->Subscribe(btmTrade, "XBTUSD");
void OnBitmexMessage(Sender: TObject; const aTopic: TwsBitmexTopics; const aMessage: string)
{
    // here you will receive the trade updates
}
```


Bitmex | 如何下单

Bitmex REST API 提供公有和私有端点。私有端点要求对消息进行签名以提高交易安全性。首先您必须登录 Bitmex 账户并创建新的 API，您将获得以下值：

- ApiKey
- ApiSecret

这些字段必须在 Bitmex API 客户端组件的 Bitmex 属性中配置。配置完成后，您可以开始向 Bitmex REST API 发起私有请求。

订单类型

所有订单都需要交易对。除非另有说明，其他所有字段均为可选。以下是有效的 ordTypes：

- **Limit**：默认订单类型。指定 orderQty 和 price。
- **Market**：传统市价单。市价单将持续执行直至成交完毕，或达到强平价格时自动取消。
- **Stop**：止损市价单。指定 orderQty 和 stopPx。当触及 stopPx 时，订单将进入挂单簿。
 - 卖单触发条件：触发价格低于 stopPx 时触发。买单则相反，高于时触发。
 - 注意：止损单在触发前不占用保证金。请确保账户中有足够的保证金，以便订单可以完全触发。
 - 平仓止损单不需要填写 orderQty。请参阅下方执行说明。
- **StopLimit**：类似于止损市价单，但下的是限价单而非市价单。指定 orderQty、stopPx 和 price。
- **MarketIfTouched**：类似于 Stop，但触发方向相反。适用于止盈订单。
- **LimitIfTouched**：如上；用于止盈限价订单。
- **挂钩单**：挂钩单允许用户提交相对于当前市场价格的限价。请指定 pegPriceType 和 pegOffsetValue。
 - 挂钩单的 execInst 必须为 Fixed，这意味着限价在订单被接受时确定，不会随参考价格变化。
 - PrimaryPeg：价格相对于近端价格设定。
 - MarketPeg：价格相对于远端价格设定。
 - 提交带有 pegPriceType 但不带 ordType 的订单将被视为挂钩单。

执行指令

支持以下 execInst。如果使用多个，请用逗号分隔（例如 LastPrice,Close）。

- **ParticipateDoNotInitiate**：也称为只挂单订单。如果此订单在下单时就会立即成交，则将取消该订单。这旨在防止在订单传输过程中远端价格向您移动。它不适用于在提交后投机远端价格移动——我们认为此类行为是滥用行为并对其进行监控。
- **MarkPrice, LastPrice, IndexPrice**：用于止损单和 if-touched 订单以确定触发价格。仅使用其中一个。默认情况下使用 MarkPrice。
- **ReduceOnly**：仅减仓订单只能减少您的仓位，不能增加。如果您的仅减仓限价单在订单簿中挂单期间仓位被其他订单减少，则其订单数量将被调减或取消。若存在多个仅减仓订单，则最不激进的将先被调整。
- **Close**：Close 意味着 ReduceOnly。Close 订单将取消具有相同方向和交易对且未结数量超过当前持仓的其他活跃限价单。这对止损很有用：通过取消这些订单，Close 止损可确保有执行所需的保证金，并且只能执行到您持仓的完整大小。如果未指定 orderQty，Close 订单的 orderQty 等于您的当前持仓大小。
 - 请注意，没有 orderQty 的 Close 订单需要 side，以便 BitMEX 知道是否应在 stopPx 上方或下方触发。
- **LastWithinMark**：由使用 LastPrice 的止损订单使用，仅允许在以下情况下触发止损：
 - 对于卖出止损市价/止损限价订单
 - 最新价 <= 止损价
 - 最新价 >= 标记价格 × (1 - 5%)
 - 对于买入止损市价/止损限价订单：
 - 最新价 >= 止损价
 - 最新价 <= 标记价格 × (1 + 5%)
- **Fixed**：钉住订单的 execInst 必须为 Fixed。这意味着限价在订单被接受时确定，并且不会随参考价格的变化而变化。

挂价订单

挂单允许用户相对于当前市场价格提交限价。限价在提交时设定一次，不随参考价格变化。此订单类型不适用于在提交后投机远端价格的移动——我们认为此类行为属于滥用并对其进行监控。

挂钩订单的 `ordType` 为 `Pegged`，`execInst` 为 `Fixed`。

还必须提交 `pegPriceType` 和 `pegOffsetValue`：

- `PrimaryPeg` - 价格相对于近端触价设置
- `MarketPeg` - 价格相对于远端买卖价差设定

追踪止损挂单

使用 `TrailingStopPeg` 的 `pegPriceType` 创建追踪止损。

价格在提交时确定，若底层价格（最新价/标记价/指数价）变动超过 0.1%，则每秒更新一次。当市场远离挂钩方向移动时，`stopPx` 随之移动；当市场向挂钩方向移动时，`stopPx` 冻结。

使用 `pegOffsetValue` 设置订单的 `stopPx`。挂单设置为 `execInst` 中指定的触发价格（默认 `MarkPrice`）。对于止损卖出和触价买入订单，请使用负偏移量。

需要 `ordType`：`Stop`、`StopLimit`、`MarketIfTouched`、`LimitIfTouched`。

追踪止损

您可以使用 '`TrailingStopPeg`' 的 `pegPriceType` 创建追踪止损单。当市场远离锚定价时，锚定的 `stopPx` 会随之移动；当市场向锚定价靠近时，则会冻结。

使用时，结合 `pegOffsetValue` 设置订单的 `stopPx`。挂钩设置为 `execInst` 中指定的触发价格（默认为"`MarkPrice`"）。对于止损卖出和触价买入订单，请使用负偏移量。

需要 `ordType`：'`Stop`'、'`StopLimit`'、'`MarketIfTouched`'、'`LimitIfTouched`'。

追踪您的订单

如果您想自行跟踪订单 ID，请为每个订单设置唯一的 `clOrdID`。此 `clOrdID` 将作为订单及任何相关执行（包括 `WebSocket` 上）的属性返回，可用于获取或取消订单。最大长度为 36 个字符。

示例：

```
// 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 是全球最大、最先进的加密货币交易平台之一。用户可以交易比特币、以太坊、瑞波币、EOS、比特币现金、Iota、NEO、莱特币、以太坊经典等。

Bitfinex WebSocket API 版本为 2.0

通过 Bitfinex WebSocket 频道发送和接收的每条消息均以 JSON 格式编码

交易对可以是交易对或保证金货币：

- 交易对符号的组成方式是在货币对前加上字母 "t"（例如 tBTCUSD、tETHUSD）。
- 保证金货币符号在货币前加前缀 "f"（例如 fUSD、fBTC.....）

连接成功后，触发 **OnBitfinexConnect** 事件，并以 Bitfinex API 版本号作为参数。

您可以调用 **Ping** 方法来测试与服务器的连接。

如果服务器发送任何信息，可通过 **OnBitfinexInfoMessage** 事件处理，其中 Code 和 Message 是包含服务器发送消息信息的参数。示例代码：

```
20051：停止/重启 WebSocket 服务器（请重新连接）
20060：正在进入维护模式。请 暂停所有活动，并在收到信息消息 20061 后恢复（最多需要 120 秒）。
20061：维护已结束，您可以恢复正常活动。建议重新取消订阅并订阅所有频道。
```

如果发生错误，将引发 **OnBitfinexError**，并提供错误信息。示例错误代码：

```
10000：未知事件
10001：未知交易对
```

要更改配置，请调用 **Configuration** 方法，并传入以下标志之一作为参数：

```
CS_DEC_S = 8; // Enable all decimal as strings.
CS_TIME_S = 32; // 将所有时间启用为日期字符串。
CS_SEQ_ALL = 65536; // 启用排序 测试功能
CHECKSUM = 131072; // 对每次订单簿迭代启用校验和。检查订单簿每一侧的前 25 条。校验和为有符号整数。
```

订阅公共频道

部分通道是公开的，无需对服务器进行认证。所有消息均通过 **OnBitfinexUpdate** 事件触发。

SubscribeTicker

行情是对市场状态的高层概览，显示当前最优买卖价及最新成交价，并包含日成交量及过去一天的价格变动等信息。

```
// 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

每当 Bitfinex 发生交易时，该频道就会发送一条成交消息，包含成交的所有相关详情，如价格、数量和时间。

```

// on trading pairs (ex. tBTCUSD)
[
    CHANNEL_ID,
    [
        [
            ID,
            MTS,
            AMOUNT,
            PRICE
        ],
        ...
    ]
]
// on funding currencies (ex. fUSD)
[
    CHANNEL_ID,
    [
        [
            ID,
            MTS,
            AMOUNT,
            RATE,
            PERIOD
        ],
        ...
    ]
]

```

SubscribeOrderBook

Order Books 频道允许您跟踪 Bitfinex 订单簿的状态。它以价格聚合为基础提供，精度可自定义。收到响应后，您将收到订单簿的快照，随后在订单簿发生任何变化时收到更新。

```

// on trading pairs (ex. tBTCUSD)
[
    CHANNEL_ID,
    [
        [

```

```

        PRICE,
        COUNT,
        AMOUNT
    ],
    ...
]
]

// on funding currencies (ex. fUSD)
[
    CHANNEL_ID,
    [
        [
            RATE,
            PERIOD,
            COUNT,
            AMOUNT
        ],
        ...
    ]
]
]
```

SubscribeRawOrderBook

这些是粒度最细的订单簿。

```

// on trading pairs (ex. tBTCUSD)
[
    CHANNEL_ID,
    [
        [
            ORDER_ID,
            PRICE,
            AMOUNT
        ],
        ...
    ]
]

// on funding currencies (ex. fUSD)
[
    CHANNEL_ID,
    [
        [
            OFFER_ID,
            PERIOD,
            RATE,
            AMOUNT
        ],
        ...
    ]
]
]
```

SubscribeCandles

提供访问图表蜡烛图信息的方式。时间框架：

1m：一分钟
 5m：五分钟
 15m：15 分钟
 30m：30 分钟
 1h：一小时
 3h：3 小时
 6h：6 小时
 12h：12小时
 1D：一天
 7D：一周
 14D：两周
 1M：一个月

```
[
  CHANNEL_ID,
  [
    [
      MTS,
      OPEN,
      CLOSE,
      HIGH,
      LOW,
      VOLUME
    ],
    ...
  ]
]
```

订阅已认证频道

此频道允许您实时了解账户状态。您可以接收关于持仓、余额、订单和交易的更新。

使用 **Authenticate** 方法对服务器进行身份验证并设置所需参数。

认证成功后，您将收到以下内容的更新：订单、仓位、成交、资金报价、资金信用、资金贷款、钱包、余额信息、保证金信息、资金信息、资金成交.....

如果您想从服务器注销，可以调用 **UnAuthenticate** 方法。

API Kucoin

Kucoin

Kucoin 是一家国际多语言加密货币交易所。它提供一些 API 以访问 Kucoin 数据。以下 API 受支持：

1. **WebSocket 流**：允许您订阅某些方法并实时获取数据。服务器将事件推送给订阅者。使用 WebSocket 作为协议。
2. **REST API**：客户端可向服务器请求市场和账户数据。需要 API Key、Secret 和 Passphrase 进行认证，并使用 HTTPs 作为协议。

属性

Kucoin API 有两种方法：公开方法和私有方法。公开方法无需身份验证即可访问，例如获取行情价格。与用户数据相关的私有方法需要使用 Kucoin API 密钥。

- **ApiKey**：您可以在 KuCoin 账户中申请新的 API Key，只需将值复制到此属性即可。
- **ApiSecret**：API 密钥仅在 REST API 中需要，WebSocket API 中只有部分方法需要 ApiKey。
- **Passphrase**：连接到 Kucoin 服务器所需的字符串。
- **Sandbox**：如果启用，将连接到 KuCoin 演示账户（默认为 false）。
 - **HTTPLogOptions**：在文本文件中存储 HTTP 请求日志
 - **Enabled**：如果启用，将存储 WebSocket API 的所有 HTTP 请求。
 - **FileName**：存储日志的文件的完整路径
 - **REST**：在文本文件中存储 REST API 请求日志
 - **Enabled**：如果启用，将存储 REST API 的所有 HTTP 请求。
 - **FileName**：存储日志的文件的完整路径。

最常见的用途

- **WebSockets API**
 - [如何连接到 WebSocket API](#)
 - [如何订阅 WebSocket 频道](#)
- **REST API**
 - [如何获取市场数据](#)
 - [如何使用私有 REST API](#)
 - [如何进行现货交易](#)
 - [私有请求时间](#)

WebSocket Feed

要从特定服务器订阅通道消息，客户端应向服务器发送订阅消息。如果订阅成功，当响应设置为 true 时，系统将向您发送 ack 消息。

```
{
  "id":"1545910660739",
  "type":"ack"
}
```

当有主题消息产生时，系统将向客户端发送相应的消息。

支持以下订阅/取消订阅方法。

公共频道

方法	参数	描述
SubscribeSymbolTicker	交易对	订阅此主题可获取 BBO 变动推送。若一秒内无变动则不推送。每 100 毫秒推送一次最新 BBO，若与上次数据相比无变化则不推送。
SubscribeAllSymbolsTicker		订阅此主题，获取所有市场交易对最优买卖价（BBO）变动的推送。
SubscribeSymbolSnapshot	交易对	订阅以获取单个交易对的快照数据。快照数据每 2 秒推送一次。
SubscribeMarketSnapshot	市场	订阅此主题以获取整个市场的快照数据。快照数据每 2 秒推送一次。
SubscribeLevel2MarketData	交易对	订阅此主题以获取 Level2 订单簿数据。WebSocket 订阅成功后，系统将通过 WebSocket 实时推送增量变更数据。
SubscribeLevel2_5BestAskBid	交易对	系统将每 100 毫秒返回一次 5 档最优买卖报价的快照数据（即实时每 100 毫秒返回 5 档最优买卖报价数据）。
SubscribeLevel2_50BestAskBid	交易对	系统将返回 50 个最优买卖报价数据，这是每 100 毫秒一次的快照数据（即每 100 毫秒实时返回 50 个最优买卖报价数据）。
SubscribeKlines	交易对	订阅此主题以获取 K 线数据。
SubscribeMatchExecutionData	交易对	订阅此主题以获取 Level 3 的撮合事件数据流。每次订单成交时，系统将以以下格式向您发送撮合消息。
SubscribeIndexPrice	交易对	订阅此主题以获取保证金交易的指数价格。
SubscribeMarkPrice	交易对	订阅此主题以获取保证金交易的标记价格。
SubscribeOrderBookChanged	交易对	订阅此主题以获取保证金交易的订单簿变化。
SubscribeLevel1	交易对	订阅某个交易对的 Level 1 最优买卖价数据。

如果将 ACK 参数设为 true，在成功订阅/取消订阅后，客户端将收到相关通知消息。

私有频道

需要从您的 Kucoin 账户获取有效的 ApiKey。ApiKey、ApiSecret 和 Passphrase 必须在客户端 API 组件的 Kucoin 属性中设置。

每次发生变化时，以下数据将被推送给客户端。如果设置了有效的 ApiKey，无需订阅任何方法，系统将自动完成订阅。

方法	描述
SubscribeTradeOrders	本主题将推送您的订单的所有变更事件。
SubscribeAccountBalance	当账户余额发生变化时，您将收到此消息。消息包含变化的详情。
SubscribePositionStatus	当持仓状态发生变化时，系统将推送变更事件。
SubscribeMarginTradeOrders	当订单进入订单簿时，系统将向出借方推送此消息。
SubscribeStopOrder	当系统收到止损订单时，您将收到类型为 "open" 的消息，表示该订单已进入系统并等待触发。
SubscribeTradeOrdersV2	订阅 V2 交易订单频道，获取增强的订单更新通知。
SubscribeCrossMarginPosition	订阅全仓保证金持仓更新。当全仓保证金持仓发生变化时，系统将推送变更事件。
SubscribeIsolatedMarginPosition	订阅逐仓保证金持仓更新。当逐仓保证金持仓发生变化时，系统将推送变更事件。

REST API

所有端点均返回 JSON 对象或数组。

公共 API 端点

这些端点无需任何授权即可访问。

通用端点

方法	参数	描述
GetServiceStatus		测试与 REST API 的连接并获取服务状态
GetServerTime		测试与 Rest API 的连接性并获取当前服务器时间。

市场数据端点

方法	参数	描述
GetSymbolList	市场	通过此端点请求获取可交易的货币对列表。若要获取交易品种的市场信息
GetTicker	交易对	通过此端点请求获取一级市场数据。返回值包括最优买价和买量、最优卖价和卖量，以及最新成交价和最新成交量。
GetAllTickers		请求市场中所有交易对的行情数据（包括 24 小时交易量）。
Get24hrStats	交易对	通过此端点请求，获取指定行情在最近 24 小时内的统计数据。
GetMarketList		通过此端点请求获取整个交易市场的交易货币。
GetPartOrderBook20	交易对	通过此端点请求，获取某个交易对的未平仓订单列表。Level-2 订单簿包含所有买卖报价（按价格聚合），此级别仅返回每个活跃价格的一个数量（仿佛该价格只有一个订单）。系统将在订单簿中返回 20 条数据（买入和卖出数据）。
GetPartOrderBook100	交易对	通过此端点请求获取某个交易对的未完成订单列表。二级订单簿包含所有买卖挂单（按价格聚合），此级别对每个活跃价格只返回一个数量（如同该价格只有一笔订单）。系统将返回订单簿中的 100 条数据（买卖双方各 50 条）。
GetFullOrderBook	交易对	通过此端点请求，可获取指定交易对的订单簿。Level 2 订单簿包含所有买单和卖单（按价格聚合）。该级别对每个价格仅返回一个聚合数量（即视为该价格只有一个订单）。此 API 将返回完整深度的数据。
GetKLines	交易对	通过此端点请求获取指定标的的 K 线数据。数据根据请求的类型按分组桶返回。
GetCurrencies		通过此端点请求获取货币列表。
GetCurrencyDetail	货币	通过此端点获取指定货币的货币详情
GetFiatPrice		通过此端点获取指定货币的货币详情
GetPartOrderBook1	交易对	通过此端点获取某个交易对的一级最优买卖价。

私有 API 端点

需要 APIKey 和 APISecret 才能获得服务器授权。

用户端点

方法	参数	描述
GetAllSubAccounts		您可以通过此接口获取所有子用户的用户信息。
GetListAccounts		获取账户列表。
GetAccount	AccountId	单个账户的信息。当您知道 accountId 时, 请使用此端点。
GetAccountBalanceSubAccount	SubUserId	此端点返回由 subUserId 指定的子用户的账户信息。
InnerTransfer		此 API 端点可用于在账户之间内部划转资金。用户可在主账户、交易账户、全仓保证金账户和逐仓保证金账户之间免费划转资金。支持从主账户、全仓保证金账户和交易账户向期货账户划转, 但不支持从期货账户向其他账户划转。
GetDepositAddresses	货币	获取货币的存款地址。
CreateDepositAddress	货币	为某种货币创建新的存款地址。
GetDepositList		获取充值历史记录。
GetAccountLedgers		获取账户账本条目。
GetTradeFees	交易对	获取指定交易对的交易手续费。

提现端点

方法	参数	描述
GetWithdrawalsList		获取提款列表。
GetHistoricalWithdrawalsList		KuCoin V1 历史提现记录列表。
GetWithdrawalsQuotas	货币	获取提款限额
ApplyWithdraw	Currency, Address, Amount	创建提款
CancelWithdraw	WithdrawalId	只有 PROCESSING 状态的提现请求才能被取消。

交易端点

方法	参数	描述
PlaceOrder		您可以下两种类型的订单：限价单和市价单。只有在账户有足够资金的情况下才能下单。订单下达后, 账户资金将在订单有效期间被冻结。冻结多少资金以及哪些资金取决于订单类型和指定的参数
PlaceMarketOrder		下市价单。
PlaceLimitOrder		下一个限价单。
PlaceMarginOrder		下达保证金订单。
CancelOrder		通过订单 ID 取消订单。
CancelOrderByClientOid		按客户订单 ID 取消订单。
CancelAllOrders		取消所有未成交订单。

ListOrders		通过此端点请求获取当前订单列表。项目分页显示，并按最新优先排序
GetRecentOrders		请求此端点以获取过去 24 小时内的 1000 笔订单。
GetOrder		通过此端点请求，按订单 ID 获取单个订单信息。
GetOrderByClientOid		通过此端点请求，按客户端订单 ID 获取单个订单信息。
ListFills		通过此端点请求获取最近的成交记录。
GetRecentFills		通过此端点请求获取最近 24 小时内的 1000 条成交记录列表。
PlaceStopOrder		下止损单。
PlaceStopMarketOrder		下达止损市价订单。
PlaceStopLimitOrder		下止损限价单。
CancelStopOrder		按订单 ID 取消未触发的止损订单
CancelStopOrderByClientOid		按客户端订单 ID 取消未触发的止损订单
CancelAllStopOrders		取消所有止损单
GetStopOrder		通过此接口请求，通过订单 ID 获取止损订单信息。
GetStopOrderByClientOid		通过此接口，使用客户订单 ID 请求获取止损订单信息。
ListStopOrders		通过此端点请求获取您当前未触发的止损订单列表。条目已分页并按最新时间排序。
PlaceHFOOrder		下高频订单。
CancelHFOOrder		按订单 ID 取消高频订单。
CancelHFOOrderByClientOid		通过客户端订单 ID 取消高频订单。
CancelAllHFOOrders		取消所有高频订单。
GetHFActiveOrders		获取活跃的高频订单。
GetHFDoneOrders		获取已完成的高频订单。
GetHFOOrder		通过订单 ID 获取特定高频订单。

事件

Kucoin 消息在 TsgcWebSocketClient 组件中接收，您可以使用以下事件：

OnConnect

成功连接到 Kucoin 服务器后。

OnDisconnect

从 KuCoin 服务器断开连接后

OnMessage

服务器发送给客户端的消息在此事件中处理。

OnError

如果协议出现任何错误，将调用此事件。

OnException

如果存在未处理的异常，将调用此事件。

此外，Kucoin API 组件中有一个名为 **OnKucoinHTTPException** 的特定事件，每次调用 HTTP 请求（REST API 或 WebSocket Feeds）出错时都会触发。

Kucoin | 连接 WebSocket API

要连接到 Kucoin WebSocket API，只需创建一个新的 Kucoin API 客户端并将其附加到 TsgcWebSocketClient 即可。

请参见下方示例：

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin();
oKucoin->Client = oClient;
oClient->Active = true;
```

Kucoin | Subscribe WebSocket Channel

Kucoin 提供多种频道，可订阅以获取市场数据、订单等实时更新。以下是订阅 Ticker 的示例：

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin();
oKucoin->Client = oClient;
oKucoin->SubscribeSymbolTicker("BTC-USDT");
void OnMessage(TsgcWSConnection *Connection, const string aText)
{
    // here you will receive the ticker updates
}
```

Kucoin | 获取市场数据

Kucoin 通过 REST 端点提供公开市场数据，当您调用这些端点之一时，您将获得所请求市场数据的快照。市场数据端点不需要身份验证，所有用户均可免费访问。

示例：要获取 BTC-USDT 行情的快照，请进行以下调用

```
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin(this);
ShowMessage(oKucoin->REST_API->GetTicker("
BTC-USDT
"));
```

Kucoin | 私有 REST API

Kucoin REST API 提供公开和私有端点。私有端点要求对消息进行签名，以提高交易安全性。首先，您必须登录您的 Kucoin 账户并创建一个新的 API，您将获得以下值：

- ApiKey
- ApiSecret
- 密码短语

这些字段必须在 Kucoin API 客户端组件的 Kucoin 属性中配置。配置完成后，即可开始向 Kucoin Pro REST API 发送私有请求

*私有请求要求您的本地计算机时间已同步，否则请求将被 Kucoin 服务器拒绝。请查阅以下关于此问题的文章：[Kucoin 私有请求时间](#)。

```
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin(this);
oKucoin->Kucoin->ApiKey = "<your api key>";
oKucoin->Kucoin->ApiSecret = "<your api secret>";
oKucoin->Kucoin->Passphrase = "<your passphrase>";
ShowMessage(oKucoin->REST_API->GetListAccounts());
```

Kucoin | 现货交易

Kucoin 允许您通过其 REST API 进行现货交易。

配置

首先，您必须在 Kucoin 账户中创建一个 **API Key**，并为现货交易添加相应权限。

完成此操作后，您可以开始现货交易。

首先，在 **Kucoin 客户端组件**中设置您的 **ApiKey**、**ApiSecret** 和 **Passphrase**，这将用于对发送到 Kucoin 服务器的请求进行签名。

下达订单

要下新订单，只需调用 Kucoin 客户端组件的 **REST_API.PlaceOrder** 方法。

根据订单类型（市价、限价等），API 所需的字段多少有所不同。

参数

参数	type	描述
clientOid	String	用户创建的唯一订单 id，用于标识订单，例如 UUID。
side	String	买入 或 卖出
symbol	String	有效的交易品种代码，例如 ETH-BTC
type	String	[可选] limit 或 market （默认为 limit ）
备注	String	[可选] 订单备注，长度不得超过 100 个 utf8 字符。
stp	String	[可选] 自成交预防， CN 、 CO 、 CB 或 DC
tradeType	String	[可选] 交易类型： TRADE （现货交易）， MARGIN_TRADE （杠杆交易）。默认值为 TRADE 。注意：为提升系统性能、加快下单和处理速度，KuCoin 已新增杠杆下单接口。仍在当前接口的交易者请尽快迁移至新接口。当前接口将于 2021 年 5 月 1 日（UTC）起不再接受杠杆订单。届时，KuCoin 将通过公告通知用户，请关注。

限价单参数

参数	type	描述
price	String	基础货币的单价
size	String	要买入或卖出的基础货币数量
timeInForce	String	[可选] GTC , GTT , IOC , 或 FOK （默认值为 GTC ），请阅读 有效时间 。
cancelAfter	long	[可选] 在 n 秒后取消，需要 timeInForce 为 GTT
postOnly	boolean	[可选] 仅挂单标志，当 timeInForce 为 IOC 或 FOK 时无效
hidden	boolean	[可选] 订单不会显示在订单簿中
iceberg	boolean	[可选] 订单簿中仅显示订单的一部分
visibleSize	String	[可选] 冰山订单的最大可见数量

市价单参数

参数	type	描述
size	String	[可选] 以基础货币计价的所需金额
funds	String	[可选] 希望使用的报价货币数量。

下订单时，有 2 种可能性：

1. **成功**：函数 PlaceOrder 返回 Kucoin 服务器发送的消息。
2. **错误**：异常将在 OnKucointHTTPException 事件中返回。

下市价单 1 BTC-USDT

```
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin(this);
oKucoin->Kucoin->ApiKey = "<api key>";
oKucoin->Kucoin->ApiSecret = "<api secret>";
oKucoin->Kucoin->Passphrase = "<passphrase>";
ShowMessage(oKucoin->REST_API->

PlaceMarketOrder

(
    kosBuy, "BTC-USDT", 1
));
```

以 40000 下限价单购买 1 BTC-USDT

```
TsgcWSAPI_Kucoin *oKucoin = new TsgcWSAPI_Kucoin(this);
oKucoin->Kucoin->ApiKey = "<api key>";
oKucoin->Kucoin->ApiSecret = "<api secret>";
oKucoin->Kucoin->Passphrase = "<passphrase>";
ShowMessage(oKucoin->REST_API->

PlaceLimitOrder

(
    kosBuy, "BTC-USDT", 1, 40000
));
```

Kucoin | 私有请求时间

当您向 Kucoin 发出私有请求时，消息会被签名以提高请求的安全性。消息中包含本地时间，如果本地时间与 Kucoin 服务器的差异超过 5 秒，请求将被拒绝。因此，验证您的本地时间已同步非常重要，您可以使用操作系统的时间同步方法来完成此操作。

您可以通过调用方法 **GetServerTime** 检查 Kucoin 服务器时间，该方法将返回 Kucoin 服务器的时间。

API Kucoin Futures

Kucoin Futures

Kucoin 是一家国际多语言加密货币交易所。它提供一些 API 以访问 Kucoin 数据。以下 API 受支持：

1. **WebSocket 流**：允许您订阅某些方法并实时获取数据。服务器将事件推送给订阅者。使用 WebSocket 作为协议。
2. **REST API**：客户端可向服务器请求市场和账户数据。需要 API Key、Secret 和 Passphrase 进行认证，并使用 HTTPs 作为协议。

属性

Kucoin API 有 2 种类型的方法：公共方法和私有方法。公共方法无需身份验证即可访问，例如获取 Ticker 价格。与用户数据相关的私有方法需要使用 Kucoin API 密钥。

- **ApiKey**：您可以在 KuCoin 账户中申请新的 API Key，只需将值复制到此属性即可。
- **ApiSecret**：API 密钥仅在 REST API 中需要，WebSocket API 中只有部分方法需要 ApiKey。
- **Passphrase**：连接到 Kucoin 服务器所需的字符串。
- **Sandbox**：如果启用，将连接到 KuCoin 演示账户（默认为 false）。
 - **HTTPLogOptions**：在文本文件中存储 HTTP 请求日志
 - **Enabled**：如果启用，将存储 WebSocket API 的所有 HTTP 请求。
 - **FileName**：存储日志的文件的完整路径
 - **REST**：在文本文件中存储 REST API 请求日志
 - **Enabled**：如果启用，将存储 REST API 的所有 HTTP 请求。
 - **FileName**：存储日志的文件的完整路径。

最常见的用途

- **WebSockets API**
 - [如何连接到 WebSocket API](#)
 - [如何订阅 WebSocket 频道](#)
- **REST API**
 - [如何获取市场数据](#)
 - [如何使用 Private REST API](#)
 - [如何进行期货交易](#)
 - [私有请求时间](#)

WebSocket Feed

要从特定服务器订阅通道消息，客户端应向服务器发送订阅消息。如果订阅成功，当响应设置为 true 时，系统将向您发送 ack 消息。

```
{
  "id": "1545910660739",
  "type": "ack"
}
```

当有主题消息产生时，系统将向客户端发送相应的消息。

支持以下订阅/取消订阅方法。

公共频道

方法	参数	描述
SubscribeSymbolTickerV2	交易对	订阅本主题以获取 BBO 变化的实时推送。订阅后，当订单簿发生变化时，系统将向您实时推送行情标的信息。建议使用新主题以获取及时信息。
SubscribeSymbolTicker	交易对	订阅此主题以获取 BBO 变化的实时推送。行情频道在每次撮合时提供实时价格更新。如果同时撮合多个订单，只推送最后一次撮合事件。
SubscribeLevel2MarketData	交易对	订阅此主题以获取 Level 2 订单簿数据。
SubscribeExecutionData	交易对	对于每笔已执行的订单，系统将以如下格式向您发送成交消息。
SubscribeLevel2_5BestAskBid	交易对	最多每 100 毫秒返回一次。
SubscribeLevel2_50BestAskBid	交易对	最多每 100 毫秒返回一次。
SubscribeContractMarketData	交易对	订阅此主题以获取合约的市场数据。
SubscribeSystemAnnouncements	交易对	订阅此主题以获取系统公告。
SubscribeTransactionStatistics	交易对	交易统计数据将每 5 秒推送给用户。
SubscribeKlines	交易对	订阅合约 K 线（蜡烛图）数据。
SubscribeFundingFeeSettlement	交易对	订阅资金费用结算通知。

如果将 ACK 参数设为 true，在成功订阅/取消订阅后，客户端将收到相关通知消息。

私有频道

需要从您的 Kucoin 账户获取有效的 ApiKey。ApiKey、ApiSecret 和 Passphrase 必须在客户端 API 组件的 Kucoin 属性中设置。

每次发生变化时，以下数据将被推送给客户端。如果设置了有效的 ApiKey，无需订阅任何方法，系统将自动完成订阅。

方法	描述
SubscribeTradeOrders	本主题将推送您的订单的所有变更事件。
SubscribeAccountBalance	当账户余额发生变化时，您将收到此消息。消息包含变化的详情。
SubscribePositionChange	当持仓状态发生变化时，系统将推送变更事件。
SubscribeStopOrder	当系统收到止损订单时，您将收到类型为 "open" 的消息，表示该订单已进入系统并等待触发。
SubscribeMarginMode	订阅保证金模式变更。当保证金模式更新时，系统将推送变更事件。
SubscribeCrossMarginLeverage	订阅全仓杠杆变更。当全仓杠杆更新时，系统将推送变更事件。

REST API

所有端点均返回 JSON 对象或数组。

公共 API 端点

这些端点无需任何授权即可访问。

通用端点

方法	参数	描述

GetServiceStatus		测试与 REST API 的连接并获取服务状态
GetServerTime		测试与 Rest API 的连接性并获取当前服务器时间。

市场数据端点

方法	参数	描述
GetOpenContractList		提交请求以获取所有未结合约的信息。
GetOrderInfoContract		提交请求以获取指定合约的信息。
GetTicker	交易对	实时行情包含最新成交价、最新成交量、交易 ID、流动性方、最优买卖价格和数量，以及订单的成交时间。这些消息也可通过 Websocket 获取。序列号用于判断 Websocket 推送的消息是否连续。
GetPartOrderBook20	交易对	获取某个交易对的聚合未平仓订单快照。
GetPartOrderBook100	交易对	获取某个交易对的聚合未平仓订单快照。
GetFullOrderBook	交易对	获取某个交易对的聚合未平仓订单快照。
GetLevel2PullingMessages	交易对	如果 WebSocket 推送的消息不连续，您可以提交以下请求并重新拉取数据，以确保序列号不缺失。在请求中，start 参数是您最后收到消息的序列号加 1，end 参数是您当前收到消息的序列号减 1。重新拉取消息并将其应用到本地交易所订单簿后，您可以继续通过 WebSocket 增量订阅来更新订单簿。如果 end 与 start 参数之差超过 500，请停止使用此请求，建议您重新构建 Level 2 订单簿。
GetTradeHistory	交易对	列出某交易对的最近 100 笔成交。
GetInterestRateList	交易对	查询利率列表。
GetIndexList	交易对	检查索引列表
GetCurrentMarkPrice	交易对	检查当前标记价格。
GetPremiumIndex	交易对	提交获取溢价指数的请求。
GetCurrentFundingRate	交易对	提交请求以检查当前标记价格。
GetKLine	交易对	获取合约 K 线数据

私有 API 端点

需要 APIKey 和 APISecret 才能获得服务器授权。

用户端点

方法	参数	描述
GetAccountOverview		获取账户概览
GetTransactionHistory		如果存在未平仓头寸，则返回的第一页状态为 Pending，表示当前 8 小时结算周期内的已实现盈亏。请在 offset 字段中指定当前页的最小偏移量以翻页。

交易端点

方法	参数	描述
PlaceOrder		您可以下两种类型的订单：限价单和市价单。仅当账户有足够资金时才能下单。订单提交后，相应资金将在订单有效期间被冻结。冻结资金的数量取决于订单类型和指定的参数。
PlaceMarketOrder		下市价单。
PlaceLimitOrder		下一个限价单。
CancelOrder		通过订单 ID 取消订单。
LimitOrderMassCancellation		取消所有未结订单（不包括止损订单）。响应是已取消订单的 orderID 列表。
StopOrderMassCancellation		取消所有未触发的止损单。响应为已取消止损单的订单 ID 列表。若要取消已触发的止损单，请使用"限价单批量撤销"。
GetOrderList		列出您当前的订单。
GetUntriggeredStopOrderList		获取未触发的止损订单列表。
GetListOrdersCompleted24hr		获取最近 24 小时内的 1000 条最新订单列表。如果您需要以低延迟获取最近的已成交订单历史，可以查询此端点。
GetOrder		按订单 ID 获取单个订单（包括止损订单）。
GetOrderByClientOid		通过客户端订单 id 获取单个订单（包括止损单）。
GetFills		获取最近成交记录列表。
GetRecentFills		获取过去 24 小时内最近 1000 条成交记录。如果您需要以低延迟获取最近的成交订单历史记录，可以查询此端点。
ActiveOrderValueCalculation		您可以查询此端点以获取所有活跃订单的总数量和总价值。
GetPositionDetails		获取指定仓位的详细信息。
GetPositionList		获取指定仓位的详细信息。
AutoDepositMargin		启用/禁用自动存入保证金
AddMarginManually		手动添加保证金
ObtainFuturesRiskLimitLevel		此接口可用于获取特定合约的风险限额信息
AdjustRiskLimitLevel		此接口用于调整风险限额等级。调整等级将取消未结订单，响应仅能指示调整请求的提交是否成功。
GetFundingHistory		提交请求以获取资金费率历史。
GetMaxOpenSize		获取合约的最大可开仓大小。
SwitchMarginMode		在全仓保证金和逐仓保证金模式之间切换。
GetMarginMode		获取合约的当前保证金模式。

事件

Kucoin 消息在 TsgcWebSocketClient 组件中接收，您可以使用以下事件：

OnConnect

成功连接到 Kucoin 服务器后。

OnDisconnect

从 KuCoin 服务器断开连接后

OnMessage

服务器发送给客户端的消息在此事件中处理。

OnError

如果协议出现任何错误，将调用此事件。

OnException

如果存在未处理的异常，将调用此事件。

此外，Kucoin API 组件中有一个名为 **OnKucoinHTTPException** 的特定事件，每次调用 HTTP 请求（REST API 或 WebSocket Feeds）出错时都会触发。

Kucoin | 期货连接 WebSocket API

要连接到 Kucoin WebSocket API，只需创建一个新的 Kucoin API 客户端并将其附加到 TsgcWebSocketClient 即可。

请参见下方示例：

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();  
TsgcWSAPI_Kucoin_Futures  
*oKucoin = new  
TsgcWSAPI_Kucoin_Futures  
();  
oKucoin->Client = oClient;  
oClient->Active = true;
```


Kucoin | Futures 订阅 WebSocket 频道

Kucoin 提供多种频道，可订阅以获取市场数据、订单等实时更新。以下是订阅 Ticker 的示例：

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kucoin_Futures
*oKucoin = new
TsgcWSAPI_Kucoin_Futures
();
oKucoin->Client = oClient;
oKucoin->SubscribeSymbolTickerV2("XBTUSD");
void OnMessage(TsgcWSConnection *Connection, const string aText)
{
    // here you will receive the ticker updates
}
```

Kucoin | Futures 获取市场数据

Kucoin 通过 REST 端点提供公开市场数据，当您调用这些端点之一时，您将获得所请求市场数据的快照。市场数据端点不需要身份验证，所有用户均可免费访问。

示例：要获取 BTC-USDT 行情的快照，请进行以下调用

```
TsgcWSAPI_Kucoin_Futures
*oKucoin = new
TsgcWSAPI_Kucoin_Futures
(this);
ShowMessage(oKucoin->REST_API->GetTicker("
XBTUSDM
"));
```

Kucoin | 期货私有 REST API

Kucoin REST API 提供公开和私有端点。私有端点要求对消息进行签名，以提高交易安全性。首先，您必须登录您的 Kucoin 账户并创建一个新的 API，您将获得以下值：

- ApiKey
- ApiSecret
- 密码短语

这些字段必须在 Kucoin API 客户端组件的 Kucoin 属性中配置。配置完成后，即可开始向 Kucoin Pro REST API 发送私有请求

*私有请求要求您的本地机器时间已同步，否则请求将被 Kucoin 服务器拒绝。请查看以下关于此问题的文章：[Kucoin 私有请求时间](#)。

```
TsgcWSAPI_Kucoin_Futures  
  
*oKucoin = new  
  
TsgcWSAPI_Kucoin_Futures  
  
(this);  
oKucoin->Kucoin->ApiKey = "<your api key>";  
oKucoin->Kucoin->ApiSecret = "<your api secret>";  
oKucoin->Kucoin->Passphrase = "<your passphrase>";  
ShowMessage(oKucoin->REST_API->GetAccountOverview());
```

Kucoin | 期货交易

Kucoin 允许您通过其 REST API 进行期货交易。

配置

首先，您必须在 Kucoin 账户中创建一个 **API Key** 并添加期货交易权限。

完成此操作后，您即可开始期货交易。

首先，在 **Kucoin 客户端组件**中设置您的 **ApiKey**、**ApiSecret** 和 **Passphrase**，这将用于对发送到 Kucoin 服务器的请求进行签名。

下达订单

要下新订单，只需调用 Kucoin 客户端组件的 **REST_API.PlaceOrder** 方法。

根据订单类型（市价、限价等），API 所需的字段多少有所不同。

参数

参数	type	描述
clientOid	String	用户创建的唯一订单 ID，用于标识其订单，例如 UUID，仅允许数字、字符、下划线(_)和分隔符(-)
side	String	buy 或 sell
symbol	String	有效的合约代码，例如 XBTUSD
type	String	[可选], limit 或 market 二选一
杠杆	String	订单杠杆倍数
备注	String	[可选] 订单备注，长度不得超过 100 个 UTF-8 字符
stop	String	[可选] down 或 up 之一。需要定义 stopPrice 和 stopPriceType
stopPriceType	String	[可选] TP 、 IP 或 MP 之一，如果指定了 stop 则必须定义。
stopPrice	String	[可选] 如果指定了 stop ，则需要定义此项。
reduceOnly	boolean	[可选] 标记以仅减少持仓量。默认设为 false。当 reduceOnly 为 true 时，需要设置持仓量。
closeOrder	boolean	[可选] 用于平仓的标记。默认设置为 false。当 closeOrder 为 true 时，将平掉所有持仓。
forceHold	boolean	[可选] 一个标记，用于强制为订单保留资金，即使该订单旨在减少持仓规模。此标记有助于订单保留在订单簿中，避免在持仓规模变化时被取消。默认为 false。

限价单参数

参数	type	描述
price	String	限价
size	Integer	订单大小，必须为正数
timeInForce	String	[可选] GTC 、 IOC （默认为 GTC ），请参阅 Time In Force
postOnly	boolean	[可选] 仅挂单标志，当 timeInForce 为 IOC 时无效。选择 postOnly 时，不允许选择隐藏或冰山。
hidden	boolean	[可选] 订单不显示在订单簿中。选择隐藏时，不允许同时选择 postOnly。
iceberg	boolean	[可选] 订单簿中仅显示订单的可见部分。选择冰山订单时，不允许选择 postOnly。
visibleSize	Integer	[可选] 冰山委托单的最大可见数量

市价单参数

参数	type	描述
size	Integer	[可选] 买入或卖出的合约数量

下订单时，有 2 种可能性：

1. **成功**：函数 PlaceOrder 返回 Kucoin 服务器发送的消息。
2. **错误**：异常将在 OnKucointHTTPException 事件中返回。

以市价单买入 1 XBTUSD

```

TsgcWSAPI_Kucoin_Futures
*oKucoint = new
TsgcWSAPI_Kucoin_Futures
(this);
oKucoint->Kucoin->ApiKey = "<api key>";
oKucoint->Kucoin->ApiSecret = "<api secret>";
oKucoint->Kucoin->Passphrase = "<passphrase>";
ShowMessage(oKucoint->REST_API->
PlaceMarketOrder
(
kosBuy, "XBTUSD", 1
));

```

在 40000 处下 1 XBTUSD 限价单

```

TsgcWSAPI_Kucoin_Futures
*oKucoint = new
TsgcWSAPI_Kucoin_Futures

```

```
(this);
oKucoin->Kucoin->ApiKey = "<api key>";
oKucoin->Kucoin->ApiSecret = "<api secret>";
oKucoin->Kucoin->Passphrase = "<passphrase>";
ShowMessage(oKucoin->REST_API->
PlaceLimitOrder
(
kosBuy, "XBTUSD", 1, 40000
));
```

Kucoin | 期货私有请求时间

当您向 Kucoin 发出私有请求时，消息会被签名以提高请求的安全性。消息中包含本地时间，如果本地时间与 Kucoin 服务器的差异超过 5 秒，请求将被拒绝。因此，验证您的本地时间已同步非常重要，您可以使用操作系统的时间同步方法来完成此操作。

您可以通过调用方法 **GetServerTime** 检查 Kucoin 服务器时间，该方法将返回 Kucoin 服务器的时间。

API 3Commas

3Commas

支持的 API

- [WebSockets API](#)：连接到公共 WebSocket 服务器并提供实时市场数据更新。
- [REST API](#)：REST API 提供账户和订单管理以及公共市场数据的端点。

WebSockets API

WebSocket 数据流提供交易和成交的实时市场数据更新

您可以订阅以下公共频道：

方法	参数	描述
SubscribeSmartTrades		
SubscribeDeals		

这些通道需要针对 3Commas 服务器进行 **身份验证**。因此，请先在您的 3Commas 账户中申请 API 密钥，然后在组件的 ThreeCommas 属性中设置相应的值：

- ApiKey
- ApiSecret

如果订阅成功，将调用事件 **OnThreeCommasConfirmSubscription**。如果失败，将调用事件 **OnThreeCommasRejectSubscription**，您可以使用 **aRawMessage** 参数获取拒绝原因。

REST API

测试连通性

方法	参数	描述
GetPing		
GetServerTime		返回服务器时间

账户

方法	参数	描述
GetAccounts		用户已连接的

		交易所列表
GetMarketList		支持的市场列表
GetMarketPairs	aMarketCode : 市场代码	所有市场交易对
GetCurrencyRatesWithLeverageData	aMarketCode : 市场代码 aPair : 交易对名称	带杠杆数据的货币汇率和限额
GetCurrencyRates	aMarketCode : 市场代码 aPair : 交易对名称	汇率和限额
GetBalances	aAccountId : 账户 ID	指定交易所的负载均衡
GetAccountTableData	aAccountId : 账户 ID	指定交易所上所

		有 用 户 余 额 的 信 息
GetAccountLeverage	aAccountId ：账 户 ID； aPair ： 交易对名称。	账 户 杠 杆 信 息
GetAccountInfo	aAccountId ：账 户 ID	单 账 户 信 息

智能交易

方法	参数	描述
GetSmartTradeHistory		获取交易历史
PlaceMarketOrder	aAccountId ：账 户 ID aOrderSide ： 买入或卖出 aPair ：交易对名 称 aQuantity ：数 量	下市价单
PlaceLimitOrder	aAccountId ：账 户 ID。 aOrderSide ：买 入或卖出。 aPair ：交易对名 称。 aQuantity ：数 量。 aPrice ：限 价。	下限价单
GetSmartTrade	ald ：交易 ID	通过交易 ID 获取智能交 易
CancelSmartTrade	ald ：交易 ID	通过交易 Id 取消智能交 易
CloseByMarketSmartTrade	ald ：交易 ID	
EditSmartTrade	ald ：交易 ID	编辑现有智 能交易

ForceStartSmartTrade	ald ：交易 ID	强制启动智能交易
AddFundsSmartTrade	ald ：交易 ID	向智能交易添加资金
GetSmartTradeTrades	ald ：交易 ID	获取智能交易的成交记录

DCA 机器人

方法	参数	描述
CreateDCABot		创建新的 DCA 机器人
GetDCABot	ald ：机器人的 id	通过 Id 获取 DCA Bot
GetDCABots		获取所有 DCA 机器人
EnableDCABot	ald ：机器人的 id	启用 DCA 机器人
DisableDCABot	ald ：机器人的 id	禁用 DCA Bot
DeleteDCABot	ald ：机器人的 id	删除 DCA 机器人
CancelDCABot	ald ：机器人的 id	取消 DCA Bot
GetDCABotStats		获取 DCA 机器人统计信息
GetAvailableStrategyList		获取可用策略列表
GetBlacklistPairs		获取黑名单交易对
AddBlacklistPairs		添加黑名单交易对

成交

方法	参数	描述
GetDeals		获取所有交易
GetDeal	ald ：交易的 ID	通过 Id 获取交易
UpdateDeal	ald ：交易的 ID	更新一笔交易
CancelDeal	ald ：交易的 ID	取消交易
CloseAtMarketDeal	ald ：交易的 ID	以市价平仓

网格机器人

方法	参数	描述
CreateGridBot		创建新的网格机器人
GetGridBot	aid ：机器人的 id	通过 Id 获取 Grid Bot
GetGridBots		获取所有网格机器人
EnableGridBot	aid ：机器人的 id	启用网格机器人
DisableGridBot	aid ：机器人的 id	禁用网格机器人
DeleteGridBot	aid ：机器人的 id	删除网格机器人

事件

OnConnect

当新的 WebSocket 连接打开时

OnDisconnect

当 WebSocket 连接关闭时

OnThreeCommasConnect

当客户端收到来自 3Commas 服务器的 Welcome 消息时，表示连接已就绪。

OnThreeCommasConfirmSubscription

确认客户端之前发送的订阅。

OnThreeCommasRejectSubscription

尝试订阅 3Commas 频道时出错

OnThreeCommasMessage

此处客户端接收服务器发送的与已订阅频道相关的数据。

OnThreeCommasPing

服务器向客户端发送 Ping。

OnThreeCommasHTTPException

如果调用 HTTP REST 方法时出现任何错误，将调用此事件。

API OKX

OKX

支持的 API

- **WebSockets API**：连接到 WebSocket 服务器，提供实时行情数据更新、账户变动及下单功能。

属性

WebSocket 频道分为两类：公开频道和私有频道。

- **公共频道**：包括行情频道、K 线频道、限价频道、订单簿频道和标记价格频道等，无需登录。
- **私有频道**：包括账户频道、订单频道和仓位频道等，需要登录。

您可以在 OKX 属性中配置以下属性。

- **ApiKey**：您可以在 OKX 账户中申请新的 API 密钥，只需将值复制到此属性。
- **ApiSecret**：API 的密钥值。
- **Passphrase**：创建新 API 密钥时定义的自定义字符串。
- **IsDemo**：如果启用，将连接到 OKX 演示账户（默认禁用）。
- **IsPrivate**：如果启用，您将能够连接到私有频道（默认禁用）。
- **IsBusiness**：如果启用，将连接到 Business WebSocket 端点（wss://ws.okx.com:8443/ws/v5/business）。用于 K 线频道、算法订单频道及其他高级频道（默认禁用）。

连接

当客户端成功连接到 OKX 服务器时，将触发 **OnOKXConnect** 事件。如果连接时出现任何错误，将触发 **OnOKXError** 事件并附带错误详情。

OnOKXConnect 事件触发后，您即可开始向 OKX 服务器发送和接收消息。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_OKX *oOKX = new TsgcWSAPI_OKX(NULL);
oOKX->Client = oClient;
oOKX->OKX->ApiKey = "alsdj23kandfnasbdfdkjhdsf";
oOKX->OKX->ApiSecret = "aldskjfk3jkadknfajndsxfj23j";
oOKX->OKX->Passphrase = "secret_passphrase";
oClient->Active = true;
void OnOKXConnect(TObject *Sender, string aMessage, string aCode, string aRawMessage)
{
    DoLog("#OKX Connected");
}
void OnOKXError(TObject *Sender, string aCode, string aMessage, string aRawMessage)
{
    DoLog("#error: " + aMessage);
}
```

公共频道

websocket 数据流提供订单和交易的实时市场数据更新。websocket 数据流有一些公共频道，如 ticker、trades 等。

您可以订阅以下公共频道：

方法	描述

SubscribeInstruments	首次订阅后将推送完整的合约列表。此后，当合约状态发生任何变化（如期货交割、期权行权、新合约/交易对上市、交易暂停等）时，将推送相关合约信息。
SubscribeTicker	检索最新成交价、买价、卖价和 24 小时交易量。数据每 100 毫秒推送一次。
SubscribeOpenInterest	检索未平仓合约数量。数据每 3 秒推送一次。
SubscribeCandlestick	检索某个合约的K线数据，推送频率最快为 500ms 推送一次数据。
SubscribeTrades	获取近期成交数据。每当有成交发生时，数据将被推送。
SubscribeEstimatedPrices	检索期货合约和期权的预计交割/行权价格。 仅在交割/行权前一小时推送预计交割/行权价格，有价格变动时推送。
SubscribeMarkPrice	检索标记价格。当标记价格发生变化时，每 200 毫秒推送一次数据；当标记价格未发生变化时，每 10 秒推送一次数据。
SubscribeMarkPriceCandlestick	检索标记价格的 K 线数据。数据每 500 毫秒推送一次。
SubscribePriceLimit	获取仪器的最高买入价和最低卖出价。当限价有变化时，数据将每 5 秒推送一次；如果限价无变化，则不推送。
SubscribeOrderBook	检索订单簿数据。 books 用于 400 档深度，book5 用于 5 档深度，bbo-tbt 用于逐笔 1 档深度，books50-l2-tbt 用于逐笔 50 档深度，books-l2-tbt 用于逐笔 400 档深度。 • books：初始完整快照中将推送 400 个深度级别。当订单簿发生变化时，每 100 毫秒推送一次增量数据。 • books5：每次推送 5 个深度级别。当订单簿发生变化时，每 100 毫秒推送一次数据。 • bbo-tbt：每次推送 1 个深度级别。当订单簿发生变化时，每 10 毫秒推送一次数据。 • books-l2-tbt：初始完整快照中将推送 400 个深度级别。当订单簿发生变化时，每 10 毫秒推送一次增量数据。 • books50-l2-tbt：初始完整快照中将推送 50 个深度级别。当订单簿发生变化时，每 10 毫秒推送一次增量数据。如果 asks 或 bids 为空数组，则表示 400 个深度级别发生了变化，而不是 50 个深度级别。如果您在本地维护订单簿数据，请忽略空的 asks 和 bids。
SubscribeOptionSummary	检索所有 OPTION 合约的详细定价信息。数据将一次性推送。
SubscribeFundingRate	检索资金费率。数据将在 30 秒至 90 秒内推送。
SubscribeIndexCandlestick	获取指数的 K 线数据。每 500 毫秒推送一次数据。
SubscribeIndexTicker	获取指数行情数据
SubscribeStatus	获取系统维护状态，并在系统维护状态变化时推送。首次订阅："推送最新变更数据"；每次状态变化时，推送变更内容。
SubscribePublicStructureBlockTrades	每当发生大宗交易时，数据将被推送。
SubscribeBlockTickers	检索最近 24 小时内的最新大宗交易量。数据将在交易执行事件触发时推送。此外，还将根据订阅粒度每 5 分钟推送一次。
SubscribeAllTrades	检索所有交易数据。每当发生交易时，数据将被推送。
SubscribeLiquidationOrders	获取强制平仓订单。当发生强制平仓订单时，将推送数据。
SubscribeADLWarning	检索 ADL 预警数据。当自动减仓风险增加时，将推送数据。
SubscribeEconomicCalendar	检索经济日历事件。当经济事件有更新时，数据将被推送。
SubscribePublicBlockTrades	获取公开大宗交易数据。每当发生大宗交易时，数据将被推送。
SubscribeOptionTrades	检索期权交易数据。每当发生期权交易时，数据将被推送。
SubscribeCallAuctionDetails	获取集合竞价详情。当集合竞价信息更新时，数据将被推送。

```

TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_OKX *oOKX = new TsgcWSAPI_OKX(NULL);
oOKX->Client = oClient;
oOKX->OKX->ApiKey = "alsdk23kandfnasbdfdkjhdsf";
oOKX->OKX->ApiSecret = "aldskjfk3jkadknfajndsfj23j";

```

```
oOKX->OKX->Passphrase = "secret_passphrase";
oClient->Active = true;
void OnOKXConnect(TObject *Sender, string aMessage, string aCode, string aRawMessage)
{
    oOKX->SubscribeInstruments(okxInstruments);
}
```

私有频道

包括账户频道、订单频道和仓位频道等，需要登录。

您可以订阅以下私有频道：

方法	描述
SubscribeAccount	获取账户信息。当下单、撤单、成交等事件触发时，数据将被推送。同时也会按照订阅粒度定期推送。
SubscribePositions	获取持仓信息。初始快照将根据订阅粒度推送。当下单/撤单等事件触发时，数据将被推送，也会根据订阅粒度定期推送。
SubscribeBalanceAndPosition	获取账户余额和持仓信息。当发生成交、资金划转等事件时，数据将被推送。
SubscribeOrders	检索订单信息。首次订阅时不会推送数据；仅在下单/撤单等事件触发时才推送数据。
SubscribeOrdersAlgo	获取算法订单（包括触发订单、OCO 订单、条件订单）。首次订阅时不推送数据，仅在下单/撤单等事件触发时才推送数据。
SubscribeAdvanceAlgo	检索高级算法订单（包括冰山订单、TWAP 订单、追踪订单）。首次订阅时推送数据，下单/撤单等事件触发时也会推送数据。
SubscribePositionRisk	此推送频道仅用作风险警告，不建议作为策略交易的风险判断依据。 在市场波动不剧烈的情况下，有可能发生在推送此消息的同时，持仓已被强制平仓的情况。
SubscribeAccountGreeks	检索账户希腊值信息。当持仓增减或账户中的现金余额发生变化等事件触发时，数据将被推送；同时也会按照订阅粒度定期推送。
SubscribeRfq	检索 Rfq 列表。
SubscribeQuotes	获取报价。
SubscribePrivateStructureBlockTrades	检索结构化大宗交易。
SubscribeSpotGridAlgoOrders	检索现货网格算法订单。首次订阅时会推送数据。当下单/撤单等事件触发时会推送数据。
SubscribeContactGridAlgoOrders	获取合约网格算法订单。首次订阅时会推送数据，下单/撤单等事件触发时也会推送数据。
SubscribeGridPositions	获取网格持仓。首次订阅时会推送数据，下单/撤单等事件触发时也会推送数据。
SubscribeGridSubOrders	获取网格子订单。首次订阅时会推送数据，下单等事件触发时也会推送数据。
SubscribeFills	检索已成交订单数据。当订单成交时，数据将被推送。
SubscribeDepositInfo	检索存款信息。当存款状态更新时，数据将被推送。
SubscribeWithdrawalInfo	检索提款信息。当提款状态更新时，数据将被推送。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_OKX *oOKX = new TsgcWSAPI_OKX(NULL);
oOKX->Client = oClient;
oOKX->OKX->ApiKey = "alsdkj23kandfnasbdfdkjhsdf";
oOKX->OKX->ApiSecret = "aldskjfk3jkadknfajndsfj23j";
oOKX->OKX->Passphrase = "secret_passphrase";
oClient->Active = true;
```

```
void OnOKXConnect(TObject *Sender, string aMessage, string aCode, string aRawMessage)
{
    oOKX->SubscribeOrders(okx Futures, "BTC-USD", "BTC-USD-200329");
}
```

交易

WebSocket 交易需要身份验证。

只有在资金充足的情况下才能下单。以下是请求参数表：

参数	类型	必填	描述
id	String	是	消息的唯一标识符 由客户端提供。它将在响应消息中返回，用于标识对应的请求。 由大小写字母数字组合、纯数字或纯字母组成，最长 32 个字符。
> instId	String	是	工具 ID，例如 <code>BTC-USD-190927-5000-C</code>
> tdMode	String	是	交易模式 保证金模式 <code>isolated</code> <code>cross</code> 非保证金模式 <code>cash</code>
> ccy	String	否	保证金货币 仅适用于 <code>cross</code> <code>MARGIN</code> 中的 <code>Single-currency margin</code> 订单。
> clOrdId	String	否	客户端提供的订单 ID 由区分大小写的字母数字组合、纯数字或纯字母组成，最多 32 个字符。
> 标签	String	否	订单标签 由区分大小写的字母数字字符组合而成，全数字或全字母，长度不超过 16 个字符。
> 侧	String	是	订单方向， <code>buy</code> <code>sell</code>
> posSide	String	可选	持仓方向 在 <code>net</code> 模式下默认为 <code>net</code> 在 <code>long/short</code> 模式下为必填项，只能为 <code>long</code> 或 <code>short</code> 。 仅适用于 <code>FUTURES/SWAP</code> 。
> ordType	String	是	订单类型 <code>market</code> ：市价单 <code>limit</code> ：限价单 <code>post_only</code> ：只挂单 <code>fok</code> ：全部成交或取消订单 <code>ioc</code> ：立即成交或取消订单 <code>optimal_limit_ioc</code> ：带立即成交或取消的市价单
> sz	String	是	买入或卖出的数量。
> px	String	可选	价格 仅适用于 <code>limit</code> 、 <code>post_only</code> 、 <code>fok</code> 、 <code>ioc</code> 订单。
> reduceOnly	Boolean	否	是否仅减少仓位， <code>true</code> <code>false</code> ，默认为 <code>false</code> 。 仅适用于 <code>MARGIN</code> 订单，以及处于 <code>net</code> 模式下的 <code>FUTURES/SWAP</code> 订单 仅适用于 <code>Single-currency margin</code> 和 <code>Multi-currency margin</code>
> tgtCcy	String	否	数量类型 <code>base_ccy</code> ：基础货币， <code>quote_ccy</code> ：计价货币 仅适用于以市价单交易的 <code>SPOT</code> 以市价单 买入默认为 <code>quote_ccy</code> ，卖出默认为 <code>base_ccy</code>
> banAmend	Boolean	否	是否禁止修改现货订单， <code>true</code> 或 <code>false</code> ，默认为 <code>false</code> 。 当 <code>banAmend</code> 为 <code>true</code> 时，余额不足将导致下单失败。仅适用于 <code>SPOT</code> 市价单。

下单示例

只有当您拥有足够的资金时，才能下单。

```
// Place Market Order
TsgcWSAPI_OKX1->PlaceMarketOrder(okxosBuy, "ETH-BTC", 1);
// Place Limit Order
TsgcWSAPI_OKX1->PlaceLimitOrder(okxosBuy, "ETH-BTC", 1, 0.25);
```

取消订单示例

取消未完成的订单。

```
TsgcWSAPI_OKX1->CancelOrder("
ETH-BTC
", "457589362405027840");
```

修改订单

修改未完成的订单。

```
TsgcWSAPI_OKX1->AmendOrder("ETH-BTC", "457589362405027840", "", 2);
```

批量交易操作

WebSocket Trade API 还支持一次性下达、取消和修改多个订单的批量操作。这些操作需要身份验证。

方法	描述
BatchPlaceOrders	在单个请求中下多个订单。每次最多可下 20 个订单。
BatchCancelOrders	在单次请求中取消多个订单。每次最多可取消 20 个订单。
BatchAmendOrders	在单个请求中修改多个未完成的订单。每次最多可修改 20 个订单。
MassCancelOrders	批量取消特定工具类型的所有挂单。

API XTB

XTB

支持的 API

- **WebSockets API**：连接到 WebSocket 服务器，提供实时市场数据更新、账户变更以及下达交易订单的功能。

属性

WebSocket 协议允许两种请求类型：**流式命令**（接收实时更新）和**检索交易数据**（向服务器发送请求以检索某些信息）。

您可以在 XTB 属性中配置以下属性。

- **User**：标识该连接的用户名。
- **Password**：用户的密钥值。
- **Demo**：若启用，将连接到 XTB 演示账户（默认禁用）。

连接

当客户端成功连接到 XTB 服务器时，事件 **OnXTBConnect** 将被触发。如果连接过程中发生任何错误，事件 **OnXTBError** 将携带错误详情被触发。
事件 **OnXTBConnect** 触发后，您即可开始从 XTB 服务器**发送和接收消息**。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_XTB *oXTB = new TsgcWSAPI_XTB(NULL);
oXTB->Client = oClient;
oXTB->XTB->User = "user_0001";
oXTB->XTB->Password = "secret_0001";
oClient->Active = true;
void OnXTBConnect(TObject *Sender, string aStreamSessionId)
{
    DoLog("#XTB Connected");
}
void OnXTBError(TObject *Sender, string aCode, string aDescription, string aRawMessage)
{
    DoLog("#error: " + aDescription);
}
```

连接命令

方法	描述
登录	客户端应用程序必须先完成登录流程才能执行任何操作。在正确完成登录流程之前，任何功能均不可用。当客户端连接到 WebSocket 服务器且已设置 User/Password 值时，登录方法将自动调用。
注销	

流式命令

您可以订阅以下频道：

方法	描述
SubscribeBalance	允许实时获取账户指标的实际值，在系统中一有更新即可获取。
SubscribeCandles	订阅和取消订阅 API 图表 K 线。每根 K 线的时间间隔为 1 分钟，每分钟推送一根新 K 线。
SubscribeKeepAlive	订阅和取消订阅"保活"消息。API 每 3 秒发送一次新的"保活"消息。
SubscribeNews	订阅或取消订阅新闻。
SubscribeProfits	订阅和取消订阅盈亏数据。
SubscribeTickPrices	建立报价订阅，并允许您在系统中一有相关信息即可实时获取。getTickPrices 命令可对同一交易对多次调用，但对于给定交易对只会创建一个订阅。请注意，当有多条记录可用时，不保证接收顺序。
SubscribeTrades	建立用户交易状态数据的订阅，允许您实时获取系统中可用的相关信息。请注意，当有多条记录可用时，接收顺序无法保证。
SubscribeTradeStatus	允许实时获取已发送交易请求的状态（一旦系统中有状态即推送）。请注意，当有多条记录时，接收顺序不保证。
SubscribePing	定期调用此函数足以刷新系统中所有组件的内部状态。流连接在会话中没有任何命令由客户端发送时，只会产生单向网络流量。建议任何不执行其他命令的应用程序至少每 10 分钟调用一次此命令。

检索交易数据

您可以发送以下请求：

方法	描述
GetAllSymbols	返回用户可用的所有交易对数组。
GetCalendar	返回包含市场事件的日历。
GetChartLastRequest	请注意，此函数通常可以被其流式等效函数 getCandles 替代，后者是检索当前蜡烛图数据的首选方式。返回从开始日期到当前时间的图表信息。如果所选 CHART_LAST_INFO_RECORD 的周期大于 1 分钟，API 返回的最后一根蜡烛图可能在周期结束前持续变化（蜡烛图每分钟自动更新一次）。
GetChartRangeRequest	请注意，此函数通常可以替换为其流式等效项 getCandles，这是检索当前 K 线数据的首选方式。返回给定开始日期和结束日期之间的图表信息和数据。
GetCommissionDef	返回佣金和汇率计算结果。该值为预期值，因此可能并非完全精确。
GetCurrentUserData	返回有关账户货币和账户杠杆的信息。
GetIbsHistory	返回给定时间范围内的 IB 数据。
GetMarginLevel	请注意，此函数通常可以由其流式等效版本 getBalance 替代，后者是获取账户指标的首选方式。返回各种账户指标。
GetMarginTrade	返回给定金融工具和交易量的预期保证金。该值以预期保证金计算，因此可能不完全准确。
GetNews	请注意，此函数通常可以被其流式等效函数 getNews 替代，后者是检索新闻数据的首选方式。返回交易服务器在指定时间段内发送的新闻。

GetProfitCalculation	根据给定的交易数据计算预估收益。该方法仅适用于类计算器应用程序。已开仓交易的收益应从服务器获取，因为服务器的计算精度更高。
GetServerTime	返回交易服务器的当前时间
GetStepRules	返回 DMA 的步长规则列表
GetSymbol	返回用户可用的交易对信息
GetTickPrices	请注意，此函数通常可以由其流式传输等价物 <code>getTickPrices</code> 替代，这是检索报价数据的首选方式。返回给定交易对的当前报价数组，仅返回自给定时间戳以来发生变化的报价。从输出中获得的新时间戳将用作下一次调用此命令的参数。
GetTradeRecords	返回 <code>orders</code> 参数中列出的订单的交易数组
GetTrades	请注意，此函数通常可被其流式等效方法 <code>getTrades</code> 替代，后者是检索交易数据的首选方式。返回用户的交易数组。
GetTradesHistory	请注意，此函数通常可被其流式等效函数 <code>getTrades</code> 替代，后者是检索交易数据的首选方式。返回在指定时间段内已平仓的用户交易数组。
GetTradingHours	返回报价和交易时间。
GetVersion	返回当前 API 版本。
Ping	定期调用此函数足以刷新系统中所有组件的内部状态。建议任何不执行其他命令的应用程序至少每 10 分钟调用一次此命令。请注意，此函数的流式对应物是 <code>ping</code> 和 <code>getKeepAlive</code> 的组合
TradeTransaction	启动交易事务。 <code>tradeTransaction</code> 将主要交易信息发送到服务器。
TradeTransactionStatus	请注意，此函数通常可以由其流式传输等价物 <code>getTradeStatus</code> 替代，这是检索交易状态数据的首选方式。返回当前交易状态。在交易处理的任何时候，客户端都可以检查服务器端的交易状态。为此，客户端必须提供从 <code>tradeTransaction</code> 调用中获取的唯一订单号。

API Bybit

Bybit

支持的 API

- **WebSocket API**：连接到 WebSocket 服务器，提供实时市场数据更新、账户变动等信息。
- **REST API**：发送 HTTP 请求以获取市场数据、下单、账户数据等。

目前，支持的 API 版本为 V5。V5 API 为 Bybit 的产品线带来了统一性和效率，在一套规范中统一了现货、衍生品和期权。

OpenAPI 版本	账户类型	线性			反向		现货	选项
		USDT 永续	USDC 永续合约	USDC 期货	永续	期货		
V5	统一交易账户	✓	✓	✓	见注释		✓	✓
	经典账户	✓			✓	✓	✓	
V3	统一交易账户	✓	✓					✓
	经典账户	✓			✓	✓	✓	

*注意：统一账户支持反向交易。但是，使用的保证金来自反向衍生品钱包，而非统一钱包。

属性

您可以在 Bybit 属性中配置以下属性。

- **ApiKey**：您可以在 Bybit 账号中申请新的 API 密钥，将值复制到此属性。若设置了 ApiKey，客户端将连接到 WebSocket 私有服务器；若为空，则连接到 WebSocket 公共服务器。
- **ApiSecret**：API 的密钥值。
- **SignatureExpires**：签名过期前的秒数（默认为 10 秒）。
- **TestNet**：若启用，将连接到 Bybit TestNet 演示账户（默认禁用）。

连接

当客户端成功连接到 Bybit 服务器时，**OnConnect** 事件触发。**OnConnect** 事件触发后，您可以开始向 Bybit 服务器发送和接收消息。如果连接到私有 WebSocket 频道，必须等到 **OnBybitAuthentication** 事件触发且 success 参数为 true 后，再订阅任何频道。

该客户端支持多个 API，请使用 BybitClient 属性设置您要使用的 API：

- **bybSpot**
- **bybInverse**
- **bybLinear**

- bybPerpetual

以下是连接到 WebSocket Spot 私有 API 的示例。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bybit *oBybit = new TsgcWSAPI_Bybit(NULL);
oBybit->Client = oClient;
oBybit->Bybit->ApiKey = "a1sdjk23kandfnasbdfdkjhdsf";
oBybit->Bybit->ApiSecret = "a1dskjfk3jkadknfajndsxfj23j";
oBybit->BybitClient = bybSpot;
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
    DoLog("#Bybit Connected");
}
```

成功连接到现货 WebSocket 服务器后，您可以开始订阅 WebSocket 频道。只需访问 **SPOT** 属性，然后调用任意可用的订阅/取消订阅方法即可。

事件

bybit 客户端实现了以下事件，用于控制连接流程并获取从 WebSocket 服务器发送的数据：

- **OnConnect**：当 WebSocket 客户端连接到 WebSocket 服务器时触发。
- **OnDisconnect**：当 WebSocket 客户端从 WebSocket 服务器断开连接时触发。
- **OnBybitAuthentication**：当客户端对私有 WebSocket 服务器完成身份验证时触发。
- **OnBybitSubscribe**：当客户端订阅 WebSocket 通道时触发。
- **OnBybitUnSubscribe**：当客户端取消订阅 WebSocket 通道时触发。
- **OnBybitData**：当客户端从服务器接收到数据时触发。
- **OnBybitError**：Bybit WebSocket 连接期间发生任何错误时触发。
- **OnBybitHTTPException**：当 REST 请求发生任何错误时触发。

WebSocket API

websocket 数据流提供订单和交易的实时市场数据更新。websocket 数据流有一些公共频道，如 ticker、trades 等。

您可以订阅以下频道：

方法	公共或私有	描述
SubscribeOrderBook	公开	订阅订单簿流。支持不同深度。
SubscribeTrade	公开	订阅最新成交流。
SubscribeTicker	公开	订阅行情流。
SubscribeKLine	公开	订阅 K 线流。
SubscribeLiquidation	公开	订阅强制平仓流
SubscribeLT_KLine	公开	订阅杠杆代币 K 线数据流。
SubscribeLT_Ticker	公开	订阅杠杆代币行情数据流。
SubscribeLT_Nav	公开	订阅杠杆代币行情数据流。
SubscribePosition	私有	订阅杠杆代币净值数据流。
SubscribeExecution	私有	订阅
SubscribeOrder	私有	订阅
SubscribeWallet	私有	订阅
SubscribeGreek	私有	订阅
SubscribeDcp	私有	订阅
SubscribeInsurance	公开	订阅保险基金流。

SubscribeOrderPriceLimit	公开	订阅订单价格限制流。
SubscribeADLAlert	公开	订阅自动减仓预警流。
SubscribeFastExecution	私有	订阅快速执行流。

以下是成功身份验证后订阅私有 WebSocket 频道的示例。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bybit *oBybit = new TsgcWSAPI_Bybit(NULL);
oBybit->Client = oClient;
oBybit->Bybit->ApiKey = "alsdkj23kandfnasbdfdkjhdsdf";
oBybit->Bybit->ApiSecret = "aldskjfk3jkadknfajndsxfj23j";
oBybit->BybitClient = bybSpot;
oClient->Active = true;
void OnBybitAuthentication(TObject *Sender, bool aSuccess, const string aError, const string aRawMessage)
{
    if (aSuccess == true)
    {
        oClient->SubscribeOrderBook("BTCUSDT");
        oClient->SubscribeTrade("BTCUSDT");
    }
}
```

REST API

REST API 提供一系列公共和私有方法，用于请求以下数据：市场行情、私有账户和钱包。以下是可用方法的列表。

方法	公开 / 私有
GetServerTime	公开
GetKLine	公开
GetMarkPriceKLine	公开
GetIndexPriceKLine	公开
GetPremiumIndexPriceKLine	公开
GetInstrumentsInfo	公开
GetOrderBook	公开
GetTickers	公开
GetFundingRateHistory	公开
GetPublicRecentTradingHistory	公开
GetOpenInterest	公开
GetHistoricalVolatility	公开
GetInsurance	公开
GetRiskLimit	公开
GetDeliveryPrice	公开
GetLongShortRatio	公开
PlaceOrder	私有
PlaceMarketOrder	私有
PlaceLimitOrder	私有
AmendOrder	私有
CancelOrder	私有
GetOpenOrders	私有
CancelAllOrders	私有
GetOrderHistory	私有
GetPositionInfo	私有
SetLeverage	私有
SwitchCrossIsolatedMargin	私有
SetTPSLMode	私有

SwitchPositionMode	私有
SetRiskLimit	私有
SetTradingStop	私有
SetAutoAddMargin	私有
AddOrReduceMargin	私有
GetExecution	私有
GetClosedPNL	私有
ConfirmNewRiskLimit	私有
GetWalletBalance	私有
GetAccountInfo	私有
GetTransactionLog	私有
BatchPlaceOrder	私有
BatchAmendOrder	私有
BatchCancelOrder	私有
SetDCP	私有
GetFeeRate	私有
GetCollateralInfo	私有
SetMarginMode	私有
GetBorrowHistory	私有
GetCoinGreeks	私有
GetCoinInfo	私有
GetAllCoinsBalance	私有
CreateInternalTransfer	私有
GetInternalTransferList	私有
GetDepositRecords	私有
GetDepositAddress	私有
CreateWithdrawal	私有
CancelWithdrawal	私有
GetWithdrawalRecords	私有

以下是获取未成交订单的示例。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bybit *oBybit = new TsgcWSAPI_Bybit(NULL);
oBybit->Client = oClient;
oBybit->Bybit->ApiKey = "alsdk23kandfnasbdfdkjhdsf";
oBybit->Bybit->ApiSecret = "aldskjfk3jkadknfajndsrfj23j";
oBybit->BybitClient = bybSpot;
oBybit->REST_API->GetAccountInfo();
```


API Forex

[Forex.com](#) (StoneX / CityIndex)

Forex.com 是由 StoneX 运营的零售交易平台。该组件封装了 Forex.com、StoneX 和 CityIndex（三者共用同一平台）使用的 Trading REST API 和基于 Lightstreamer 的流式服务。支持以下 API：

1. **Streaming API**：Lightstreamer TLCP 2.5 通过 WebSocket 传输。成功完成 REST LogOn 后，价格报价、订单更新、持仓更新、账户保证金和报价将实时推送。
2. **REST API**：会话登录（UserName + Password + AppKey）、交易和止损限价订单管理、账户查询和市场数据。使用 HTTPS。

组件

- **TsgcWSAPI_Forex**：主客户端。保存 **Credentials**，暴露 **REST_API**、内部 **Lightstreamer** 流式传输客户端和 **OnForex*** 事件。
- **TsgcHTTP_API_Forex**：REST 组件，可通过主客户端的 **REST_API** 属性访问。
- **TsgcWSPClient_Lightstreamer**：内部 TLCP 2.5 子协议客户端，用于流式订阅（PRICES / ORDERS / QUOTES / CLIENTACCOUNTMARGIN / TRADEMARGIN）。

已发布属性

- **凭据**：登录 REST API 所需的字段组。
 - **UserName**：您的 Forex.com / StoneX / CityIndex 账户用户名。
 - **Password**：账户密码。
 - **AppKey**：交易场所颁发的应用程序密钥。
 - **AppVersion**：可选的客户端版本字符串。
 - **AppComments**：登录时记录的自由格式文本。
 - **StreamingHost**：Lightstreamer 主机（默认为公共流式传输主机）。
- **AutoReconnect**：启用后，转发到底层 WebSocket 客户端 WatchDog，以便在网络断开后自动重新连接流式会话（默认为 true）。
- **REST_API**：对 **TsgcHTTP_API_Forex** 组件的引用。
- **Lightstreamer**：引用内部 **TsgcWSPClient_Lightstreamer** 子协议客户端（AdapterSet 固定为 STREAMINGALL）。
- **SessionToken**：只读。REST LogOn 调用返回的会话令牌；用作 Lightstreamer 的 **LS_password**。
- **ClientAccountId / TradingAccountId**：只读。成功登录后填充。

已发布的事件

OnForexConnect

流媒体会话创建成功且可以开始订阅时触发。

OnForexDisconnect

流式会话结束时触发（服务器关闭、网络中断或显式断开连接）。

OnForexException

在 REST 错误（包括 LogOn 失败）和流式会话故障时触发。

OnForexPriceTick

每次 PRICES 更新都会触发：Bid、Offer、Price、Change、Direction、AuditId、TickDate。

OnForexOrderUpdate

针对每次 ORDERS 更新（新订单、已成交、已取消、已拒绝）触发。

OnForexPositionUpdate

每次活跃交易账户上的持仓更新时触发。

OnForexAccountMargin

为每次 CLIENTACCOUNTMARGIN / TRADEMARGIN 更新（权益、保证金、可用保证金、货币）触发。

OnForexQuote

每次 QUOTES 更新（报价请求工作流）时触发。

OnForexSubscribe

每次服务器确认 Lightstreamer 订阅（SUBOK / SUBCMD）时触发一次。参数：*aAdapter*（PRICES、ORDERS、QUOTES、CLIENTACCOUNTMARGIN 或 TRADEMARGIN）、*aGroup*（项目组，例如 *ID.<ClientAccountId>*）、*alters*、*aFields*。

OnForexUnsubscribe

当订阅结束时触发，原因可能是 **UnwatchMarket**、流断开或服务器发送 UNSUB 帧。参数：*aAdapter*、*aGroup*。

最常见的用途

- **REST API**
 - [如何连接 REST API](#)
 - [如何下交易和止损限价订单](#)
 - [如何查询账户、持仓和订单历史](#)
 - [如何获取市场数据](#)
- **WebSockets API**
 - [如何连接到 WebSocket API](#)
 - [如何订阅流适配器](#)

示例

创建客户端，设置凭证，连接，订阅市场价格流，然后断开连接。

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey = "<appkey>";
oForex->OnForexPriceTick = ForexPriceTick;
oForex->Connect();
oForex->WatchMarket(401484830);
oForex->Disconnect();
```

Forex | 连接 REST API

每次调用 Forex.com / StoneX / CityIndex 交易 API 都需要一个有效会话。会话状态通过 **LogOn** 获取，通过 **LogOff** 结束，并通过 **Ping** 保持活跃。

凭据

在主客户端组件上设置凭据。这些凭据将在第一次调用时复制到 REST 会话中。

- **UserName**：Forex.com/StoneX/CityIndex 账户用户名。
- **Password**：账户密码。
- **AppKey**：由交易所颁发的应用程序密钥。
- **AppVersion / AppComments**：登录时记录的可选字符串。

会话方法

方法	参数	描述
LogOn	UserName, Password, AppKey	打开 REST 会话。返回完整 JSON 响应，并将 SessionToken、ClientAccountId 和 TradingAccountId 存储在主客户端上。所有后续 REST 和流式调用均使用该令牌进行身份验证。
LogOff		使服务器上的会话失效并清除令牌。
Ping		保持会话活跃。服务器会话在一段时间不活跃后过期；当没有其他 REST 调用时，请定期发送 Ping。
GetServiceStatus		报告当前服务状态（无需身份验证）。

令牌流程

LogOn 返回的令牌以 *UserName* HTTP 标头的形式附加到后续每个 REST 请求。同一令牌在创建 Lightstreamer 会话时被流式客户端复用为 *LS_password* 字段。无需调用两次 LogOn：对主客户端调用 **Connect** 会先执行 REST LogOn，然后才打开流式套接字。

示例

打开 REST 会话并调用 Ping

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey = "<appkey>";
ShowMessage(oForex->REST_API->LogOn(oForex->Credentials->UserName,
                                     oForex->Credentials->Password,
                                     oForex->Credentials->AppKey));
ShowMessage(oForex->REST_API->Ping());
oForex->REST_API->LogOff();
```

Forex | 交易和止损限价订单

REST API 提供五个订单管理端点。每个端点均提供三种重载形式，以便调用方根据所需的控制级别进行选择。

端点

方法	描述
NewTradeOrder	在差价合约市场下市价/最优价交易订单。
UpdateTradeOrder	附加或更改现有未平仓交易上附加的止损/限价保护。
NewStopLimitOrder	下待成交止损单或限价单（尚未进入市场）。
UpdateStopLimitOrder	修改现有的待处理止损单或限价单。
CancelOrder	取消待处理的止损单或限价单。

过载层级

订单方法提供两种重载：

1. **标量参数重载**（仅限 NewTradeOrder 和 CancelOrder）— 接受最常用的字段（NewTradeOrder 的 MarketId、TradingAccountId、Direction、Quantity、BidPrice / OfferPrice、AuditId；CancelOrder 的 OrderId、TradingAccountId、MarketId）。帮助方法在内部构建请求，适用于简单的无保护订单。
2. **类型化订单对象重载**，接受 **TsgcHTTPForexTradeOrder**、**TsgcHTTPForexStopLimitOrder** 或 **TsgcHTTPForexCancelOrder**。这些对象公开了所有文档化字段，包括 **IfDone**（附加的止损/限价括号）、**Close** 数组（按订单 ID 平仓语义）、**PositionMethodId**、**Reference** 和部分成交标志。这是除简单市价单以外所有情况的推荐路径；对象构建约定与 Kucoin 等其他交易场所共享。

TsgcHTTPForexTradeOrder 字段

- **MarketId**：由 ListCfdMarkets 或 FullSearchWithTags 返回的数字市场 ID。
- **TradingAccountId**：活跃的交易账户。在 LogOn 后从主客户端读取。
- **Direction**：buy（买）或 sell（卖）。
- **Quantity**：以市场原生单位表示的交易量。
- **BidPrice / OfferPrice**：订单提交时客户端观察到的价格（Forex.com 使用这些价格检测过时报价）。
- **AuditId**：来自最新价格 Tick 的价格审计 ID。
- **PositionMethodId**：净头寸/对冲行为（1 = 净额, 2 = 对冲）。
- **IfDone**：父订单成交时激活的附加订单列表（止损、限价、保证止损、追踪止损）。
- **Close**：此订单成交时要关闭的订单 ID 数组（部分平仓）。
- **Reference**：调用方提供的客户端参考，在订单更新中回显。

示例

以市场价附带止盈和止损买入 1000 EUR/USD

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey = "<appkey>";
oForex->Connect();

TsgcHTTPForexTradeOrder *oOrder = new TsgcHTTPForexTradeOrder();
oOrder->MarketId = 401484830;
oOrder->TradingAccountId = oForex->TradingAccountId;
oOrder->Direction = fdBuy;
oOrder->Quantity = 1000;
oOrder->BidPrice = 1.09120;
oOrder->OfferPrice = 1.09130;
ShowMessage(oForex->NewTradeOrder(oOrder));
delete oOrder;
```

Forex | 账户查询

这些端点读取账户状态和订单历史记录。它们需要一个活动会话（参见 [连接 REST API](#)）。

端点

方法	参数	描述
GetClientAndTradingAccount		返回客户端账户以及与已认证用户关联的交易账户列表。
ListOpenPositions	TradingAccountId	返回交易账户上所有当前未平仓头寸。
ListActiveStopLimitOrders	TradingAccountId	返回所有待处理的止损和限价订单。
GetOrder	OrderId	通过 id 返回单个订单，包括成交历史和附加的 IfDone 订单。
ListTradeHistory	TradingAccountId, MaxResults	返回账户的交易订单历史，最新的在前。
ListStopLimitOrderHistory	TradingAccountId, MaxResults	返回账户的止损限价订单历史，最新的排在最前面。
SimulateTrade	JSON / 标量 / 订单对象	通过服务器验证器运行交易订单而不提交。适用于预检保证金和价格容忍度检查。接受与 NewTradeOrder 相同的三个重载层级。

已停用端点

官方 Forex.com / StoneX / CityIndex 参考文档仍然记录了一些实时服务器不再提供的端点。这些端点故意不在组件中公开：

- **GetSystemLookup** — 参考文档描述了一个通用查找端点，但服务器对每个已知查找键都返回 HTTP 404。请改用 **GetClientAndTradingAccount** 和 **ListCfdMarkets**。
- **/news/headlines/*** — 新闻资讯系列在参考文档中仍有列出，但在生产服务器上返回 HTTP 500。Trading API 中没有可用的替代方案；新闻通过 Web 前端提供。

示例

列出未平仓头寸和最近 50 笔交易

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey = "<appkey>";
oForex->Connect();
ShowMessage(oForex->REST_API->ListOpenPositions(oForex->TradingAccountId));
ShowMessage(oForex->REST_API->ListTradeHistory(oForex->TradingAccountId, 50));
```

Forex | 市场数据

静态市场元数据和历史价格。这些端点用于发现 **MarketId** 值，这些值随后被传入交易端点和流式适配器。

端点

方法	参数	描述
GetMarketInformation	MarketId	返回完整的市场描述符（交易对、小数位数、点差大小、保证金因子、交易时间）。
ListCfdMarkets	SearchByMarketName, ClientAccountId, MaxResults	列出名称前缀匹配的差价合约市场。
FullSearchWithTags	Query, Tags, MaxResults	跨所有市场的自由文本搜索，支持可选标签过滤（外汇、指数、股票、大宗商品）。
GetPriceBars	MarketId, Interval, Span, PriceBars	历史 OHLC K 线。Interval 为 MINUTE、HOUR、DAY、WEEK 之一；Span 为每根 K 线的间隔数量。
GetPriceTicks	MarketId, PriceTicks	市场最新报价快照（买价/卖价/中间价）。

示例

查询 EUR/USD 并读取最近 100 根 1 分钟 K 线

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey = "<appkey>";
oForex->Connect();
ShowMessage(oForex->REST_API->ListCfdMarkets("EUR/USD", 0, 10));
ShowMessage(oForex->REST_API->GetPriceBars(401484830, "MINUTE", 1, 100));
```

Forex | 连接 WebSocket API

Forex.com / StoneX / CityIndex 通过 Lightstreamer 流式传输市场数据和账户更新。该组件通过 WebSocket 使用原生 Lightstreamer TLCP 2.5 有线协议；无需 Lightstreamer 原生库。

传输

- **WebSocket 子协议**：TLCP-2.5.0.lightstreamer.com。在 *Sec-WebSocket-Protocol* 请求头中声明。
- **适配器集**：STREAMINGALL。配置于内部 **Lightstreamer** 子协议客户端；所有 Forex.com 数据适配器（PRICES、ORDERS、QUOTES、CLIENTACCOUNTMARGIN、TRADEMARGIN）均位于此适配器集下。
- **Authentication**：从 REST **LogOn** 调用获取的会话令牌在创建 Lightstreamer 会话时以 *LS_password* 形式发送，同时将 *LS_user* 设为账户用户名。

重新连接

当启用 **AutoReconnect**（默认）时，底层 WebSocket 客户端 WatchDog 在网络错误后重新打开套接字。Lightstreamer 协议本身在服务器希望客户端移动到不同 URL 或重新绑定会话时会发出 *LOOP* 帧；组件透明地处理这些帧，仅在完整会话中断时触发 **OnForexDisconnect/OnForexConnect**。

当 REST 会话过期（长时间未 Ping）时，流式服务器关闭会话。组件使用存储的凭据再次调用 **LogOn**，刷新 **SessionToken**，并使用新令牌重启 Lightstreamer 会话。

示例

打开流式会话

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey = "<appkey>";
oForex->OnForexConnect = ForexConnected;
oForex->OnForexDisconnect = ForexDisconnected;
oForex->OnForexException = ForexException;
oForex->AutoReconnect = true;
oForex->Connect();
oForex->Disconnect();
```

Forex | 订阅流式适配器

流式数据通过 *STREAMINGALL* 适配器集下的五个 Lightstreamer 数据适配器传送。每个适配器均有其自己的条目命名规则和订阅模式，快照式状态（当前价格、当前账户保证金）使用 **MERGE** 模式，事件流频道（订单更新、持仓更新、报价事件）使用 **RAW** 模式。

适配器

DataAdapter	项目	Mode	事件	字段
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, MarketId, MarketTypeId, DirectionId, Direction, Quantity, OpenPrice, CurrentPrice, Multiplier, MarginRequirement, Currency, OpenTradeEquity

注意：

- PRICES 和 CLIENTACCOUNTMARGIN 使用统一的 *ID.{numericId}* 项目形式，并为每个项目流式传输单个可变更记录（MERGE）。
- ORDERS、QUOTES 和 TRADEMARGIN 使用旧版的每适配器前缀（*ORDERS.{id}*、*QUOTES.{id}*、*TRADEMARGIN.{id}*），并以“火水管”模式流式推送事件（RAW）。每条记录承载已登录 **ClientAccountId** 的所有订单/报价/交易，记录通过 **OrderId** / **QuoteId** 自行标识。
- TRADEMARGIN 记录与旧版 CityIndex SDK 中的官方 *TradeMarginDTO* 对应。**DirectionId** 以数字代码形式到达（1 = 买入，2 = 卖出）；库还将 **Direction** 作为字符串 “buy”/“sell” 公开。**OpenPrice** 为交易入场价格（服务器字段 *PriceTakenAt*），**CurrentPrice** 为最新估值价格（*PriceCalculatedAt*），**OpenTradeEquity** 为账户货币的浮动盈亏（*OTEConverted*）。
- ORDERS 中的 **Direction** 字段是服务器发布的数值代码（1/2）；**Status** 是字符串（例如 “Accepted”、“Open”、“Cancelled”）；**OrderType** 是字符串（“Market”/“Stop”/“Limit”）。
- TRADEMARGIN 数据流不发布每笔交易的 **TradingAccountId**、**MarketName** 或每笔交易的时间戳；每条数据项包含已登录 **ClientAccountId** 的所有持仓。如需这些字段，请调用 **REST_API.ListOpenPositions**。ORDERS 数据流不发布 **TriggerPrice** 和 **Applicability**；需要时请调用 **REST_API.GetOrder**。

辅助方法

主客户端封装了 Lightstreamer SUB/UNSUB 管道。常用的有：

- **WatchMarket(aMarketId)**：订阅某个市场的 PRICES。更新通过 **OnForexPriceTick** 到达。
- **UnwatchMarket(aMarketId)**：删除订阅。
- **WatchAccount**：为已登录账户订阅 ORDERS、CLIENTACCOUNTMARGIN 和 TRADEMARGIN。更新将通过对应事件传递。
- **WatchQuotes**：为询价工作流程订阅 QUOTES 适配器。

订阅生命周期事件

主客户端触发两个生命周期事件，与底层 TLCP SUBOK / UNSUB 帧对应：

- **OnForexSubscribe(Sender, aAdapter, aGroup, altems, aFields)**：当服务器确认任何 Lightstreamer 订阅时触发。*aAdapter* 是数据适配器名称（PRICES、ORDERS、QUOTES、CLIENTACCOUNTMARGIN、TRADEMargin）；*aGroup* 是条目组（例如 *ID.<ClientAccountId>* 或 *ORDERS.<ClientAccountId>*）；*altems* 和 *aFields* 是服务器在 SUBOK/SUBCMD 帧中报告的计数。
- **OnForexUnsubscribe(Sender, aAdapter, aGroup)**：订阅结束时触发，可通过 **UnwatchMarket**、流式断连或服务器发送 UNSUB 帧触发。

示例

订阅 EUR/USD 价格行情

```
TsgcWSAPI_Forex *oForex = new TsgcWSAPI_Forex(NULL);
oForex->Credentials->UserName = "<username>";
oForex->Credentials->Password = "<password>";
oForex->Credentials->AppKey = "<appkey>";
oForex->OnForexPriceTick = ForexPriceTick;
oForex->Connect();
oForex->WatchMarket(401484830);
```

API Blockchain

Blockchain

区块链 WebSocket API 允许开发者实时接收有关新交易和区块的通知。

WebSocket 打开后，您可以订阅频道：

- **SubscribeTransactions**：订阅所有新比特币交易的通知。
- **UnsubscribeTransactions**：取消订阅所有新比特币交易的通知。
- **SubscribeAddress**：接收特定比特币地址的新交易。
- **UnSubscribeAddress**：停止接收特定比特币地址的新交易。

事务通过 **OnNewTransaction** 事件接收：

```
{
  "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": "76a914640cfd7b79d94d1c980133e3587bd6053f091f388ac"
      }
    ]
  }
}
```

- **SubscribeBlocks**：当发现新区块时接收通知。注意：如果链发生分叉，您将针对特定区块高度收到多条通知。
- **UnSubscribeBlocks**：停止在发现新区块时接收通知。注意：如果链发生分叉，您将为特定区块高度收到多条通知。

区块通过 **OnNewBlock** 事件接收：

```
{
  "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 允许接收实时通知，无需发送额外请求，是从交易所获取数据的更快方式。Cex 组件有一个名为 Cex 的属性，您可以在其中填写 Cex 提供的 API 密钥以访问您的账户数据。

消息编码

- 所有消息均以 JSON 格式编码。
- 价格以字符串形式呈现，以避免客户端 JSON 解析时出现精度误差。
- 服务器不支持 WebSocket 帧压缩。
- 时间以整数 UNIX 时间戳（秒）表示。

身份验证

要访问 CEX.IO WebSocket 数据，您需要授权。

- 登录 CEX.IO 账号。
- 前往 <https://cex.io/trade/profile#/api> 页面。
- 选择所需权限的类型。
- 点击“生成密钥”按钮并保存您的密钥，激活后将无法再次查看。
- 激活您的密钥。

连接性

- 如果已连接的套接字连续 15 秒不活跃，CEX.IO 服务器将发送 PING 消息。
- 只有服务器可以发起 PING 请求。
- 服务器仅向已验证身份的用户发送 ping。
- 用户必须以 PONG 消息响应。否则，WebSocket 将被断开连接。此操作由库自动处理。
- 对于已认证用户，若在 15 秒内未收到服务器的通知或 ping，为确保连接和认证有效，建议发送 'ticker' 或 'get-balance' 等请求并等待响应。

公共频道

这些频道不需要身份验证。来自服务器的响应通过 OnCexMessage 事件接收。

- **SubscribeTickers**：仅包含所有交易对最新成交价格的行情数据流（已弃用）

```
{
  "e": "tick",
  "data": {
    "symbol1": "BTC",
    "symbol2": "USD",
    "price": "428.0123"
  }
}
```

- **SubscribeChart**：包含开盘价、最高价、最低价、收盘价、成交量数字的 OHLCV 图表数据流（已弃用）

```
{
  'e': 'ohlcv24',
  'pair': 'BTC:USD',
  'data': [
    '418.2936',
    '420.277',
    '412.09',
    '416.9778',
    '201451078368'
  ]
}
```

- **Subscribe Pair**：市场深度推送（已弃用）

```
{
  'e': 'md_grouped',
  'data': {
    'pair': 'BTC:USD',
    'id': 11296131,
    'sell': {
      '427.5000': 1000000,
      '480.0000': 263544334,
      ...
    },
    'buy': {
      '385.0000': 3630000,
      '390.0000': 1452458642,
      ... 400+ pairs together with 'sell' pairs
    }
  }
}
```

- **订阅货币对**：订单簿数据流（已弃用）

```
{
  '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
    ]
  }
}
```

私有频道

要访问这些频道，请先调用 `Authenticate` 方法。来自服务器的响应通过 `OnCexMessage` 事件接收。

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-subscribe",
  "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

在 CEX.IO 系统中，订单可以存在于撮合引擎或归档数据库中。可能存在一段时间（约 2 秒或更长），在此期间订单已完成/已取消，但尚未转移到归档数据库。这意味着您无法通过 `archived-orders/open-orders` 调用查看它。此调用允许在任何情况下获取订单信息。响应格式可能因订单所在位置而异。

```
{
  "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

支持的 API

- [WebSockets API](#)：连接到公共 WebSocket 服务器，提供实时行情数据更新。

WebSockets API

WebSocket 是一种基于 TCP 的全双工通信协议。全双工意味着双方可以使用同一通信信道异步地互相发送消息。本节描述 Exchange Plus 与客户端之间应互相发送哪些消息。所有消息均应为有效的 JSON 对象。

WebSocket API 主要用于获取信息或执行 REST API 中不可用或不易操作的操作。但是，某些请求或操作可以通过 REST API 和 WebSocket API 两者完成。Exchange Plus 向客户端发送消息，作为对客户端之前发送的请求的响应，或作为某些事件的通知（无需客户端事先请求）。

公共 API 调用

公开 API 速率限制是为了保护系统免受 DDoS 攻击并确保所有客户端对端点享有同等稳定访问权限。请求限制由与每个公开 API 调用相关的成本决定。

Exchange Plus 将公共 API 调用限制为每分钟最多 100 积分，每次公共 API 调用均有其对应的积分成本（见下文）。如果达到请求速率限制，Exchange Plus 将返回错误，向客户端发送断开连接事件，并随后关闭 WS 连接。Exchange Plus 将从下一个整分钟开始继续为客户端提供服务。在以下示例中，请求计数器将在 11:02:00.000 时重置。

方法	描述
GetTicker	该方法旨在获取 Ticker 的当前信息，包括特定资产的当前价格、24 小时价格和成交量变化、最新成交事件等数据。
GetOrderBook	此方法允许客户端接收特定交易对的当前订单簿快照。
GetCandles	通过 Candles 方法，客户端可以接收不同分辨率和数据类型的历史 OHLCV 蜡烛图。客户端可以指定额外的时间框架和数量限制过滤条件，使响应更精确地满足客户端需求。
GetTradeHistory	此方法允许客户端获取所请求交易对的历史成交数据。客户端可以使用附加过滤参数（如时间范围、成交 ID 范围、方向等）来补充成交历史请求，以接收符合请求参数的成交记录。
GetServerTime	此方法用于获取 Exchange Plus 服务器上的当前时间。对于需要与服务器时间同步的应用程序，此方法非常有用。
GetPairsInfo	Pair Info 方法允许客户端接收所有支持交易对的参数。
GetCurrenciesInfo	Currencies Info 方法允许客户端接收 Exchange Plus 中配置的所有货币参数，以及 Exchange Plus 与 CEX.IO 钱包之间的存取款可用性信息。
GetProcessingInfo	此请求允许客户端获取关于从外部钱包充值和提现到外部钱包的可用选项的详细信息，涵盖每种支持的加密货币，包括加密货币名称及充提款可用的区块链。此外，针对每条支持的区块链，还会说明该区块链上的加密货币类型、当前的充提款可用性、最小充提款金额以及外部提款手续费。处理信息使客户在选择接收充值地址和通过特定区块链发起外部提款时更加灵活，让客户能以更便捷的方式将加密资产转入或转出 CEX.IO 生态系统。
SubscribeOrderBook	客户端通过 WebSocket 订阅可以订阅指定交易对的订单簿数据。在响应订单簿订阅请求时，客户端将收到具有指定 seqId 编号的请求交易对的当前（初始）订单簿快照。为跟踪订单簿的后续更新，客户端需要通过 WebSocket 订阅 "order_book_increment" 消息，该消息包含交易对名称、seqId 编号以及买卖价格档位的增量变化。

UnSubscribeOrderBook	从订单簿频道取消订阅。
SubscribeTrade	通过使用交易订阅方法，客户端可以通过 WebSocket 订阅请求交易对上发生的实时交易事件。作为对交易订阅请求的响应，客户端将收到一个唯一的交易订阅标识符，当不再需要该订阅时，应使用该标识符进行取消订阅。客户端应通过 WebSocket 订阅 "tradeHistorySnapshot" 和 "tradeUpdate" 消息，以接收初始和定期的交易历史快照以及所请求交易对的实时交易事件。
UnSubscribeTrade	取消订阅交易频道。

示例：获取 BTC-USD 交易对的最新 Ticker

```
TsgcWebSocketClient* oClient;
TsgcWSAPI_CexPlus* oCexPlus;

void __fastcall OnCexPlusConnectEvent(System::TObject* Sender)
{
    oCexPlus->GetTicker("BTC-USD");
}

void __fastcall OnCexPlusMessageEvent(System::TObject* Sender, UnicodeString Event, UnicodeString Msg)
{
    ShowMessage("Ticker data: " + Msg);
}

void SetupCexPlus()
{
    oClient = new TsgcWebSocketClient(NULL);
    oCexPlus = new TsgcWSAPI_CexPlus(NULL);
    oClient->Active = true;
    oCexPlus->Client = oClient;
    oCexPlus->OnConnect = OnCexPlusConnectEvent;
    oCexPlus->OnMessage = OnCexPlusMessageEvent;
}
```

私有 API 调用

Exchange Plus 使用 API 密钥来允许访问私有 API。

客户端可以通过 Exchange Plus Web 终端的 API 密钥管理配置文件部分生成、配置和管理 API 密钥，设置权限级别、API 密钥的白名单 IP 等。

API 密钥限制：默认情况下，客户端最多可拥有 5 个 API 密钥。

为了在使用 API 密钥时限制对某些功能的访问，应为每个 API 密钥定义特定的权限集。已定义的权限集可根据需要进一步编辑。

以下权限级别可用于 API 密钥：

- **读取：**查看账户相关数据、接收报告、订阅市场数据等的权限级别。
- **Trade：**权限级别，允许代表账户下单和取消订单。
- **Funds Internal：**权限级别，允许在 CEX.IO Exchange Plus Portfolio 的账户之间（子账户之间，或主账户与子账户之间）转移资金。
- **资金钱包：**权限级别，允许在 CEX.IO Exchange Plus 投资组合账户（主账户和子账户）与 CEX.IO 钱包之间相互转账。

方法	描述
GetCurrentFee	此方法指示特定时刻的当前手续费，综合考虑了客户最新的 30 天交易量及星期几（例如周末的手续费可能有所不同）。
GetFeeStrategy	手续费策略返回所有可适用于客户端的手续费选项，考虑因素包括客户端的交易量、星期几、交易对、交易对组别等。此方法提供关于总体手续费策略的信息，涵盖所有可能

	适用于客户端的交易手续费值。若要获取基于客户端当前 30 天交易量的当前交易手续费，请使用 [Current Fee] 方法。若要获取当前 30 天交易量，请使用 [Volume] 方法。
GetVolume	此请求允许客户端接收其过去 30 天的 USD 等值交易量。
CreateAccount	此请求允许客户端创建新的子账户。默认情况下，客户端最多可拥有 5 个子账户（含主账户）。
GetAccountStatus	通过使用账户状态 V3 方法，客户端可以了解当前余额及其在换算货币（默认为"USD"）中的参考等值，以及每个子账户和货币中在未结（活跃）订单中锁定的金额。如果客户端有可用的交易手续费余额，则响应还将包含一般性交易手续费余额数据，例如促销名称、货币名称、总余额以及该促销在交易手续费余额上的到期日期。客户端有责任追踪其子账户的可用交易余额，即当前子账户余额减去子账户未结（活跃）订单中锁定的余额金额。
GetOrders	此请求允许客户端查询其订单信息。
NewOrder	客户端可以通过 WebSocket API 使用"我的新订单请求"下单。随同对该请求的响应，如果请求成功，Exchange Plus 还会向客户端发送账户事件和执行报告消息。响应消息指示系统在发送响应时订单的最新状态。如果客户端未收到"我的新订单请求"的响应消息，可以使用带有 clientOrderId 参数的"获取我的订单请求"查询当前订单状态。发送新订单请求时，强烈建议使用 clientOrderId 参数，该参数对应客户端的特定新订单请求。Exchange Plus 会避免对具有相同 clientOrderId 的订单进行多次下单。如果识别到多个具有相同 clientOrderId 和其他相同订单参数的新订单，Exchange Plus 仅处理第一个订单，并在响应具有相同参数的第二个及后续新订单请求时返回该订单的状态。如果识别到多个具有相同 clientOrderId 但其他订单参数不同的新订单，Exchange Plus 仅处理第一个订单，并拒绝具有相同 clientOrderId 但其他订单参数不同的第二个及后续新订单请求。
NewMarketOrder	下一个市价单。
NewLimitOrder	下达新的限价单。
CancelOrder	客户端可以取消订单。若请求成功，交易所将向客户端发送账户事件和执行报告消息。此外，若取消订单请求被拒绝，交易所将发送订单取消拒绝消息。
CancelAllOrders	客户端可通过 WebSocket API 取消所有挂单。收到此请求后，Exchange Plus 将开始对所有挂单执行取消流程，并向客户端发送相应的账户事件和执行报告消息。
GetTransactionHistory	此请求允许客户端查询其财务交易记录（存款、提款、内部转账、佣金或交易）。
GetFundingHistory	此请求允许客户端查找其存款和提款交易记录。
InternalTransfer	客户端可以请求在其子账户之间或在其主账户与子账户之间转账。Exchange Plus 不对客户账户间的资金转移收取任何手续费。若请求成功，Exchange Plus 将同时向客户发送账户事件消息。
GetDepositAddress	此方法可供客户端用于接收用于存入加密货币的加密地址。可以为主账户和子账户生成存款地址。
FundsDepositFromWallet	客户端可以将资金从 CEX.IO 钱包存入 Exchange Plus 账户。系统会避免处理具有相同 clientTxId 的多个存款请求。如果收到多个具有相同 clientTxId 的存款请求，系统仅处理第一个存款请求，并拒绝具有相同 clientTxId 的第二个及后续存款请求。
FundsWithdrawalToWallet	客户端可以从 Exchange Plus 账户提现到 CEX.IO 钱包。系统避免使用相同 clientTxId 的多次提现请求。如果收到多个具有相同 clientTxId 的提现请求，系统只处理第一个提现请求，并拒绝具有相同 clientTxId 的第二个及后续提现请求。
GetWalletBalance	获取 CEX.IO 钱包余额信息。
SubscribeAccountEvents	订阅实时账户事件通知（余额变化、订单成交）。
UnSubscribeAccountEvents	取消订阅账户事件通知。

示例：获取订单。

```
TsgcWebSocketClient* oClient;
TsgcWSAPI_CexPlus* oCexPlus;
```

```
void __fastcall OnCexPlusAuthenticatedEvent(System::TObject* Sender)
{
    oCexPlus->GetOrders();
}

void __fastcall OnCexPlusMessageEvent(System::TObject* Sender, UnicodeString Event, UnicodeString Msg)
{
    ShowMessage("Orders: " + Msg);
}

void SetupCexPlus()
{
    oClient = new TsgcWebSocketClient(NULL);
    oCexPlus = new TsgcWSAPI_CexPlus(NULL);
    oClient->Active = true;
    oCexPlus->Client = oClient;
    oCexPlus->CexPlus->ApiKey = "your-api-key";
    oCexPlus->CexPlus->ApiSecret = "your-api-secret";
    oCexPlus->OnCexPlusAuthenticated = OnCexPlusAuthenticatedEvent;
    oCexPlus->OnMessage = OnCexPlusMessageEvent;
}
```

API Discord

Discord

注意：自 API v10 起，Discord API 域名已从 discordapp.com 更改为 discord.com。该组件已更新为使用 API 版本 10 和新域名。

Gateway 是 Discord 通过安全 WebSocket 进行实时通信的形式。客户端通过其连接的 gateway 接收事件和数据，并通过 REST API 发送数据。

授权

首先，您必须生成一个新的机器人，并复制用于通过 API 进行认证的 Bot Token。然后在 API 组件中设置此令牌。

```
TsgcWSAPI_Discord1->DiscordOptions->BotOptions->Token = "...bot token here...";
```

意图

在需要处理大量数据时，尤其是规模化时，维护有状态应用程序可能比较困难。Gateway Intents 是一种帮助您降低计算负担的系统。

向 gateway 进行标识时，您可以指定一个 intents 参数，允许您有条件地订阅预定义的"意图"，即 Discord 定义的事件组。如果您不指定某个意图，将不会收到该组中的任何 gateway 事件。有效的意图为（零值表示接收所有事件）：

```
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

心跳

心跳由组件自动处理，您无需担心。当客户端连接到服务器时，服务器发送包含心跳间隔的 HELLO 响应，组件读取响应并自动调整心跳，每 x 秒发送一次 ping。有时服务器也会向客户端发送 ping；这也由客户端自动处理。

连接就绪

连接就绪后，经服务器成功登录并授权后，会触发 **OnDiscordReady** 事件，之后您可以开始接收来自服务器的更新。

连接恢复

如果连接意外关闭，当客户端尝试重新连接时，会调用 **OnDiscordBeforeReconnect** 事件。组件会自动保存成功恢复所需的所有数据，但如需要，可以修改参数。如果您不想重新连接并希望开始新的干净会话，只需将 **Reconnect** 设为 **False**。

如果会话已恢复，则触发 **OnDiscordResumed** 事件。如果是新会话，则触发 **OnDiscordReady**。

分发事件

事件通过 **OnDiscordDispatch** 分发，因此您可以在此处读取服务器发送给客户端的事件。

```
void OnDiscordDispatch(TObject *Sender, const string aEvent, const string RawData)
{
    DoLog("#discord dispatch: " + aEvent + " " + RawData);
}
```

aEvent 参数包含事件名称。

RawData 包含完整的 JSON 消息。

HTTP 请求

为了请求 guild、用户信息或更新数据.....Discord 要求使用 HTTP 请求而非 gateway WebSocket 消息，以下是所有可用于发起 HTTP 请求的方法：

```
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;
```

示例： 获取当前用户信息

```
result = GET_Request("/users/@me");
```

服务器的示例响应：

```
{
  "id": "637423922035480852",
  "username": "test",
  "avatar": null,
  "discriminator": "5125",
  "bot": true,
  "email": null,
  "verified": true,
  "locale": "en-US",
  "mfa_enabled": false,
  "flags": 0
}
```

API | OpenAI

OpenAI Realtime API 实现低延迟、多模态交互，包括语音到语音对话体验和实时转录。

组件 **TsgcWSAPI_OpenAI** 实现了 RealTime OpenAI API。

配置

使用 **method** 属性选择 **Conversation** 或 **Transcription**，目前仅支持 Transcription 模式。

InputAudio 属性允许您自定义以下数据：

- **Language**：示例：english（值 = 'en'）。
- **Model**：将使用的模型，示例：whisper-1
- **Prompt**：可选提示，用于向模型提供指令。

OpenAI

OpenAI API 使用 API 密钥进行身份验证。请访问您的 [API 密钥](#) 页面获取您在请求中使用的 API 密钥。请记住，您的 API 密钥是保密的！请勿与他人共享或在任何客户端代码（浏览器、应用程序）中公开。生产请求必须通过您自己的后端服务器路由，并从环境变量或密钥管理服务中安全加载 API 密钥。

此 **API Key** 必须在组件的 **OpenAIOptions.ApiKey** 属性中配置。对于属于多个组织的用户，如果您的账户属于某个组织，可以在 **OpenAIOptions.Organization** 属性中设置您的组织。

配置 API 密钥后，以下是与 OpenAI API 交互的可用函数列表。

Azure

该客户端支持 Microsoft Azure OpenAI 服务，因此您也可以使用 Azure 账户与 Azure OpenAI API 进行交互。要将客户端配置为与 Azure 配合使用，请按以下步骤操作：

1. 将属性 **OpenAIOptions.Provider** 配置为 **oapvAzure**
2. 设置 **ResourceName** 和 **DeploymentId** 的值（这些值可以在您的 Azure 账户中找到）
 1. **OpenAIOptions.AzureOptions.ResourceName** = <您的资源名称>。
 2. **OpenAIOptions.AzureOptions.DeploymentId** = <您的部署 ID>。
3. 设置您 Azure 账户的 API 密钥
 1. **OpenAIOptions.ApiKey** = <azure api key>。

转录示例

以下是使用 OpenAI API 进行实时转录的示例

```
TsgcWebSocketClient *WSClient = new TsgcWebSocketClient(NULL);
TsgcAudioRecorderWave *oAudio = new TsgcAudioRecorderWave(NULL);
TsgcWSAPI_OpenAI *OpenAI = new TsgcWSAPI_OpenAI(NULL);
OpenAI->Client = WSClient;
OpenAI->AudioRecorder = oAudio;

OpenAI->OpenAIOptions->ApiKey = "your-api-key-here";
OpenAI->OpenAIOptions->Method = sgcoaimTranscription;
OpenAI->OpenAIOptions->Provider = sgcoaipOpenAI;
OpenAI->InputAudio->Language = "en";
OpenAI->InputAudio->Model = "whisper-1";

void __fastcall TForm1::OnOpenAIAudioTranscriptionCompleted(TObject *Sender, TsgcWSOpenAIConversation_Item_Comple
    Log("#transcription_completed: " + aItem->Transcript);
}
```

API MEXC

MEXC 是一家全球加密货币交易所，通过安全的 WebSocket 连接提供流式市场和账户数据。

WebSocket 现货 API 遵循 [Market Streams](#) 和 [User Data Streams](#) 文档中发布的规范。市场频道以 Protocol Buffers 帧传送，而私有流使用 JSON。**TsgcWSAPI_MEXC** 组件管理连接生命周期、认证及监听密钥维护，使应用程序可以专注于处理推送数据。

除了通过 WebSocket 进行流式传输外，该工具包还提供完整的 Spot HTTP API。**TsgcHTTP_API_MEXC_Spot** 组件（单元 **sgcHTTP_API_MEXC**）封装了所有官方 REST 端点，使应用程序可以查询市场数据或通过签名请求进行交易。

组件配置

基础 WebSocket 端点为 **wss://wbs-api.mexc.com/ws**。当 **MEXCUserDataStreams.UserStream** 启用且提供了有效的 API 密钥时，组件会自动请求监听密钥并将其附加到连接 URL 中。

在窗体上（或通过代码创建）拖放一个 **TsgcWebSocketClient** 和一个 **TsgcWSAPI_MEXC** 组件，将客户端实例分配给 API 组件，然后激活客户端以启动会话。

```
TsgcWebSocketClient *WSClient = new TsgcWebSocketClient();
TsgcWSAPI_MEXC *MEXC = new TsgcWSAPI_MEXC();
MEXC->Client = WSClient;
MEXC->MEXCAPI->ApiKey = "YOUR_KEY";
MEXC->MEXCAPI->ApiSecret = "YOUR_SECRET";
MEXC->MEXCUserDataStreams->UserStream = true;
if (WSClient->Connect()) {
    MEXC->SubscribeTrade("BTCUSD");
}
```

属性

- **Client**：对 **TsgcWebSocketClient** 传输的引用。该组件启用客户端心跳（15 秒）以保持会话活跃。
- **MEXCAPI.ApiKey / ApiSecret**：Spot API 凭据。ApiKey 用于私有频道；ApiSecret 用于签名 listen key 请求。
- **MEXCUserDataStreams.UserStream**：为 **True**（默认）时，组件在连接前获取监听密钥并每 10 分钟刷新一次。禁用它可仅使用公开数据流。
- **MEXCUserDataStreams.ListenKeyOnDisconnect**：当 WebSocket 断开连接且 listen key 处于活动状态时执行的操作：
 - **blkmxodDeleteListenKey**：通过 REST 撤销 listen key（默认）。
 - **blkmxodClearListenKey**：保留服务器端的 listen key 并清除缓存值。
 - **blkmxodDoNothing**：在下次连接时重用相同的 listen key。
- **ListenKey**：只读属性，公开由 REST API 返回的活跃监听密钥。
- **RawMessages**：若启用，传入的文本帧不会被解析，而是通过继承的 WebSocket 事件以纯 JSON 形式转发。

事件

- **OnConnect / OnDisconnect**：当底层 WebSocket 会话建立或终止时触发。
- **OnMEXCMarketStream**：为每个 Protobuf 市场帧触发。处理程序接收一个 **TsgcMEXCSpotProtoMessage**，其 **MessageType** 属性标识载荷（Trade、KLine、DiffDepth、LimitDepth、BookTicker、BookTickerBatch、MiniTicker 或 MiniTickers）。使用 **sgcWebSocket_API_MEXC_Proto** 中的辅助类来检查解码后的结构。
- **OnMEXCResponse**：接收 JSON 格式的订阅确认和 ping/pong 响应（Id、Code、Message）。

- **OnMEXCUserDataStream**：当私有账户消息到达时触发（订单、成交、余额更新）。提供频道名称和原始 JSON 负载。
- **OnMEXCException**：捕获 REST 或 WebSocket 异常（监听密钥错误、HTTP 故障、解析问题等）。

以下示例演示如何解码交易帧：

WebSocket 订阅方法

该组件为每个公开现货频道提供辅助方法。调用 **Subscribe*** 方法开始流式传输数据，调用相应的 **UnSubscribe*** 停止。

公共频道

方法	参数	描述
SubscribeTrade	Symbol, Interval (ms)	以 Protobuf publicdeals 格式传送的聚合成交执行数据。
SubscribeKline	Symbol, Interval	可配置时间间隔（1 分钟至 1 个月）的蜡烛图更新。
SubscribeDiffDepth	Symbol, Interval (ms)	用于本地订单簿维护的订单簿增量深度差异数据。
SubscribeBookDepth	Symbol, Levels	以请求的深度（默认为 5）定期快照限价订单簿。
SubscribeBookTicker	Symbol, Interval (ms)	单个交易对的最优买卖价更新。
SubscribeBookTickerBatch	UTC 标志	多个交易对的聚合最优买卖价更新。
SubscribeMiniTickers	UTC 标志	所有活跃仪器的滚动 24 小时迷你行情统计。
SubscribeMiniTicker	交易对, UTC 标志	单个品种的迷你行情。

私有频道

私有主题需要有效的 API 密钥。组件使用活跃的监听密钥进行订阅，并定期更新。

方法	描述
SubscribeAccountUpdate	账户余额和持仓变化。
SubscribeAccountDeals	已成交订单的执行报告。
SubscribeAccountOrders	订单生命周期更新（新建、已取消、已拒绝等）。

使用相应的 **UnSubscribe*** 方法终止数据流。组件自动发送 **PING** 命令并验证返回的 **PONG** 以监控连接状态。

公开 REST 端点

公开可用的 REST 方法。

方法	描述
Ping	连接检查，调用 /api/v3/ping ，当交易所回复空对象时返回 True 。
GetServerTime	从 /api/v3/time 获取当前交易所时间戳。

GetExchangeInformation	从 /api/v3/exchangeInfo 返回交易规则和交易对元数据。
GetOrderBook	从 /api/v3/depth 下载订单簿快照；设置可选的 limit 参数以控制层级数量（默认 100）。
GetTrades	从 /api/v3/trades 检索最近的公共成交，可选指定 limit （默认 100）。
GetHistoricalTrades	提供历史成交数据，支持可选的 limit 和 fromId 过滤器。
GetAggregateTrades	通过 /api/v3/aggTrades 返回压缩/聚合的成交记录，支持可选的 limit 、 fromId 、 startTime 和 endTime 过滤器。
GetKlines	从 /api/v3/klines 下载蜡烛图数据，支持时间范围和数量限制过滤。
GetAveragePrice	从 /api/v3/avgPrice 返回当前加权平均价格。
Get24hrTicker	通过 /api/v3/ticker/24hr 检索一个或所有交易对的 24 小时行情统计数据。
GetPriceTicker	使用 /api/v3/ticker/price 获取最新价格；传入空交易对可获取所有交易对的价格。
GetBookTicker	从 /api/v3/ticker/bookTicker 获取最优买卖报价，支持单一品种或全交易所请求。

私有 REST 端点

私有端点需要有效的 API 密钥和已签名的查询字符串。组件自动追加时间戳、recvWindow 和 HMAC 签名。

方法	描述
GetAccountInformation	从 /api/v3/account 返回余额和权限。
GetOpenOrders	列出当前未成交订单（可选择按交易对过滤）。
GetAllOrders	检索带有可选时间和数量过滤器的历史订单。
GetOrder	通过提供 orderId 或 origClientId 查询特定订单的状态。
GetMyTrades	列出私有交易执行记录，支持可选的 limit 、 fromId 、 startTime 和 endTime 过滤器。
NewOrder	在 /api/v3/order 下单（市价、限价、止损等），可选参数包括 timeInForce 、 quantity 、 quoteOrderQty 、 price 、 newClientId 、 stopPrice 、 icebergQty 及额外参数。
TestNewOrder	向 /api/v3/order/test 发送仅验证请求，接受与 NewOrder 相同的参数集。
CancelOrder	通过 /api/v3/order 使用 orderId 或 origClientId 取消特定订单。
CancelAllOrders	通过 /api/v3/openOrders 批量取消某个交易对的所有未成交订单。
GetSubAccounts	列出托管子账户 (/api/v3/sub-account/list)。
GetSubAccountAssets	返回特定子账户的余额。
TransferSubAccount	通过 /api/v3/sub-account/transfer 在子账户之间划转资产；提供金额，并可选择提供划转 类型 。
GetDepositAddress	通过此端点请求获取存款地址，支持可选的 network 过滤器。
GetDepositHistory	获取存款记录，支持可选的 coin 、 status 、 startTime 和 endTime 过滤条件。
GetWithdrawHistory	按 coin 、 status 和可选时间范围过滤的提款历史记录。
提款	通过 /api/v3/capital/withdraw/apply 提交提现请求，包括必填的 coin 、 address 和 amount ，以及可选的 network 、 addressTag 、 withdrawOrderId 和其他参数。
BatchOrders	在单个请求中为多个标的提交批量订单。
GetTradeFee	获取特定交易对的交易手续费率。
GetDepositHistory	获取存款记录，支持可选的 coin 、 status 、 startTime 和 endTime 过滤条件。
CancelWithdraw	通过提款 ID 取消待处理的提款请求。

CreateInternalTransfer	在账户内部转移资产。
GetTransferHistory	获取账户间内部划转的历史记录。

API MEXC Futures

MEXC 的永续和交割合约提供了专用的流式 API，详见官方 [期货 WebSocket 规范](#)。TsgcWSAPI_MEXC_Futures 组件封装了消费实时衍生品数据所需的连接、登录握手和主题管理。

MEXC 还为衍生品交易维护了一个 REST 接口。sgcHTTP_API_MEXC 中包含的 TsgcHTTP_API_MEXC_Futures 组件通过与官方端点一一对应的 HTTP 辅助方法来补充 WebSocket 数据流。

组件配置

Futures WebSocket 端点为 **wss://contract.mexc.com/edge**。消息以 JSON 对象编码，交易所要求定期发送 ping/pong 帧，这由组件自动处理。

要接收私有通知（订单和账户数据），您必须进行身份验证。在激活 WebSocket 客户端之前，请设置 **MEXCAPI.ApiKey** 和 **MEXCAPI.ApiSecret**。凭据存在时，组件将在套接字连接后立即签署登录请求（HMAC SHA256）。

```
TsgcWebSocketClient *WSClient = new TsgcWebSocketClient();
TsgcWSAPI_MEXC_Futures *Futures = new TsgcWSAPI_MEXC_Futures();
Futures->Client = WSClient;
Futures->MEXCAPI->ApiKey = "YOUR_KEY";
Futures->MEXCAPI->ApiSecret = "YOUR_SECRET";
if (WSClient->Connect()) {
    Futures->SubscribeDepth("BTC_USDT", true);
}
```

属性

- **Client**：API 组件使用的 WebSocket 传输。已启用心跳支持以保持套接字活跃。
- **MEXCAPI.ApiKey / ApiSecret**：用于签署登录请求的必填项。没有凭据时只能访问公共频道。
- **RawMessages**：启用后绕过 JSON 解析器，通过基础事件公开原始文本帧。

事件

- **OnConnect / OnDisconnect**：WebSocket 生命周期通知。
- **OnMEXCLogin**：成功通过身份验证登录后触发。
- **OnMEXCSubscribed / OnMEXCUnsubscribed**：订阅变更的确认。该事件公开 MEXC 返回的频道名称。
- **OnMEXCMessage**：将每条市场或账户更新及频道标识符一并传递。
- **OnMEXCError**：当服务器回复 **rs.error** 载荷（无效参数、身份验证失败等）时触发。
- **OnMEXCException**：报告在处理 JSON 数据或执行网络操作时引发的本地异常。

以下处理程序记录订阅确认并打印传入的深度快照：

WebSocket 订阅方法

每个辅助方法将 **订阅/取消订阅** 请求封装为官方 API 所描述的格式。使用对应的 **UnSubscribe*** 方法可取消流。

方法	参数	描述
SubscribeDeal	交易对	在合约上执行的交易（频道： deal ）。
SubscribeTickers	-	所有合约的全局行情统计。

SubscribeTicker	交易对	单个合约的行情摘要。
SubscribeDepth	Symbol, Compress	增量订单簿更新
SubscribeDepthFull	Symbol, Level	可配置深度级别数的完整深度快照（默认 20）。
SubscribeKline	Symbol, Interval	支持的时间框架（Min1、Min5、Min15、Min30、Min60、Hour4、Hour12、Day1、Week1、Month1）的 K 线数据。
SubscribeFundingRate	交易对	最新资金费率公告。
SubscribeIndexPrice	交易对	标的指数价格数据流。
SubscribeFairPrice	交易对	交易所推送的标记（公允）价格更新。

服务器确认请求时，**OnMEXCSubscribed** 事件将以通道名称（例如 **rs.sub.depth.BTC_USDT**）触发。错误通过 **OnMEXCError** 转发，包含服务器消息以供故障排查。

私有频道

私有通道需要有效的 API 密钥。提供凭据后，组件将自动进行身份验证。

方法	描述
SubscribePersonalOrder	个人订单更新。
SubscribePersonalOrderDeal	个人订单成交更新。
SubscribePersonalPosition	个人持仓更新。
SubscribePersonalPlanOrder	个人计划（触发器）订单更新。
SubscribePersonalStopOrder	个人止损订单更新。
SubscribePersonalStopPlanOrder	个人停止计划订单更新。
SubscribePersonalRiskLimit	个人风险限额更新。
SubscribePersonalADLLevel	个人 ADL（自动去杠杆）级别更新。
SubscribePersonalAsset	个人资产更新。

公开 REST 端点

公开可用的 REST 方法。

方法	描述
GetPing	通过 /api/v1/ping 检查 API 可达性。
GetServerTime	从 /api/v1/time 返回服务器时间戳。
GetContracts	提供可用合约列表（ /api/v1/contract/detail ）。
GetDepth	通过 /api/v1/contract/depth 下载订单簿深度；使用可选 limit 参数（默认 50）选择级别数。
GetDeals	通过 /api/v1/contract/deals 检索最近的交易，可选 limit 参数（默认 100）。
GetKlines	通过 /api/v1/contract/kline 返回 K 线数据，支持可选的 startTime 、 endTime 和 limit 过滤条件（默认 200）。
GetIndexPrice	获取请求交易对的底层指数价格（ /api/v1/contract/indexPrice ）。
GetFairPrice	从 /api/v1/contract/fairPrice 返回合约的公允（标记）价格。
GetFundingRate	使用 /api/v1/contract/fundingRate 报告交易对的最新资金费率。
GetAllTickers	获取所有期货合约行情及实时价格摘要。
GetFundingRateHistory	获取合约的历史资金费率记录。
GetFairPriceKline	获取合约的公允价格（标记价格）K 线/蜡烛图数据。
GetIndexPriceKline	获取某一合约的指数价格 K 线/蜡烛图数据。

私有 REST 端点

私有衍生品端点需要 API 凭据。组件会自动使用时间戳和 recvWindow 对每个请求进行签名。

方法	描述
GetAccountAssets	从 /api/v1/private/account/assets 返回保证金余额。
GetPositionList	通过 /api/v1/private/position/list 列出当前持仓，可选择按 symbol 过滤。
SetPositionLeverage	更新某交易对的杠杆，如果提供了保证金模式，也一并更新。
PlaceOrder	在 /api/v1/private/order 上下期货订单，需提供 symbol 、 side 、 positionSide 、 type 和 quantity ，以及可选的 price 、 clientOrderId 和额外参数。
CancelOrder	使用 /api/v1/private/order/cancel 通过 orderId 或 clientOrderId 取消特定订单。
CancelAllOrders	通过 /api/v1/private/order/cancel-all 取消某个交易对的所有未成交订单。
GetOpenOrders	通过 /api/v1/private/order/list/open 列出当前未成交订单，支持可选的品种过滤器。
GetOrderHistory	获取订单历史（ /api/v1/private/order/list/history ），支持可选的交易对、时间范围和条数限制参数。
GetFundingHistory	从 /api/v1/private/account/funding 返回过去的资金支付，支持可选的 startTime 、 endTime 和 limit 过滤器。
GetOpenPositions	获取所有当前持仓。
ChangeMargin	更改某持仓的保证金金额。
GetPositionMode	获取当前持仓模式（单向或对冲模式）。
ChangePositionMode	在单向和对冲模式之间更改持仓模式。
PlaceBatchOrder	在单个批量请求中下多个期货订单。
GetOrderDetail	获取特定订单的详细信息。
GetOrderDealDetails	获取特定订单的成交/交易详情。
PlaceTriggerOrder	下达在满足条件时执行的触发（计划）订单。
CancelTriggerOrder	取消特定触发订单。
CancelAllTriggerOrders	取消所有未成交的触发委托单。
GetTriggerOrders	获取触发（计划）订单列表。
GetStopOrders	获取止损单列表。
CancelStopOrder	取消特定止损单。
CancelAllStopOrders	取消所有未结止损订单。

API Bitget

Bitget

支持的 API

- **WebSocket API**：连接到 WebSocket 服务器，提供实时市场数据更新、账户变动等信息。
- **REST API**：发送 HTTP 请求以获取市场数据、下单、账户数据等。

属性

使用 **BitgetChannel** 属性选择通道类型：

- **bgcSpot**：连接到现货 WebSocket API。
- **bgcFutures**：连接到期货 WebSocket API。

您可以在 Bitget 属性中配置以下属性。

- **ApiKey**：您可以在 Bitget 账户中申请新的 API 密钥，只需将其值复制到此属性。若设置了 APIKey，客户端将连接到 WebSocket 私有服务器；若为空，则连接到 WebSocket 公共服务器。
- **ApiSecret**：API 的密钥值。
- **Passphrase**：创建新 API 密钥时定义的自定义字符串。

连接

当客户端成功连接到 Bitget 服务器时，触发 **OnConnect** 事件。触发 **OnConnect** 事件后，您可以开始向 Bitget 服务器发送和接收消息。如果您正在连接私有 WebSocket 频道，则必须等到触发 **OnBitgetAuthentication** 事件且检查 success 参数为 True 后，才能订阅任何频道。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWS_API_Bitget *oBitget = new TsgcWS_API_Bitget();
oBitget->Client = oClient;
oBitget->Bitget->ApiKey = "your_api_key";
oBitget->Bitget->ApiSecret = "your_api_secret";
oBitget->Bitget->Passphrase = "your_passphrase";
oBitget->BitgetChannel = bgcSpot;
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
    DoLog("#Bitget Connected");
}
```

事件

Bitget 客户端实现以下事件，用于控制连接流程并获取 WebSocket 服务器发送的数据：

- **OnConnect**：当 WebSocket 客户端连接到 WebSocket 服务器时触发。
- **OnDisconnect**：当 WebSocket 客户端从 WebSocket 服务器断开连接时触发。
- **OnBitgetAuthentication**：当客户端向私有 WebSocket 服务器进行身份验证时触发。
- **OnBitgetSubscribe**：当客户端订阅 WebSocket 频道时触发。
- **OnBitgetUnSubscribe**：当客户端取消订阅 WebSocket 频道时。
- **OnBitgetData**：当客户端从服务器接收到数据时。
- **OnBitgetError**：当 Bitget WebSocket 连接发生任何错误时触发。
- **OnBitgetHTTPException**：REST 请求期间发生任何错误时触发。

WebSocket API

WebSocket 数据流提供订单和交易的实时市场数据更新。

您可以订阅以下频道：

方法	公共或私有	描述
SubscribeTicker	公开	订阅某一合约的行情流。

SubscribeTrade	公开	订阅最新成交流。
SubscribeCandle	公开	订阅 K 线数据流，支持多种时间间隔（如 candle1m、candle5m、candle1H）。
SubscribeOrderBook	公开	订阅订单簿流。支持不同的深度级别。
SubscribeOrders	私有	订阅订单更新。需要身份验证。
SubscribePositions	私有	订阅持仓更新。需要身份验证。
SubscribeAccount	私有	订阅账户余额更新。需要身份验证。

以下是成功身份验证后订阅私有 WebSocket 频道的示例。

```
void OnBitgetAuthentication(TObject *Sender, bool aSuccess, const string aError, const string aRawMessage)
{
    if (aSuccess == true)
    {
        oBitget->SubscribeOrders("BTCUSDT");
        oBitget->SubscribeAccount();
    }
}
```

REST API

REST API 提供公开和私有方法，用于从市场和私人账户请求数据。通过组件的 **REST_API** 属性访问 REST API。

方法	公开 / 私有	描述
GetServerTime	公开	获取服务器时间。
GetTickers	公开	获取一个或所有交易对的行情信息。
GetOrderBook	公开	获取某个交易对的订单簿。
GetCandles	公开	获取某个交易对的蜡烛图/K 线数据。
GetRecentTrades	公开	获取某个交易对的最新成交记录。
PlaceOrder	私有	下新订单。
CancelOrder	私有	取消现有订单。
GetOpenOrders	私有	获取未结订单列表。
GetOrderDetail	私有	获取特定订单的详情。
GetOrderHistory	私有	获取历史订单。
GetAccountAssets	私有	获取账户资产信息。
GetAccountInfo	私有	获取账户信息。

以下是使用 REST API 的示例。

```
TsgcWS_API_Bitget *oBitget = new TsgcWS_API_Bitget();
oBitget->Bitget->ApiKey = "your_api_key";
oBitget->Bitget->ApiSecret = "your_api_secret";
oBitget->Bitget->Passphrase = "your_passphrase";
// Get tickers
oBitget->REST_API->GetTickers("SPOT");
// Place an order
oBitget->REST_API->PlaceOrder("BTCUSDT", "buy", "market", "normal", "0.01");
```

API GateIO

Gate.io

支持的 API

- **WebSocket API**：连接到 WebSocket 服务器，提供实时市场数据更新、账户变动等信息。
- **REST API**：发送 HTTP 请求以获取市场数据、下单、账户数据等。

属性

使用 **GateIOChannel** 属性选择频道类型：

- **giocSpot**：连接到现货 WebSocket API。
- **giocFutures**：连接到期货 WebSocket API。

您可以在 GateIO 属性中配置以下属性。

- **ApiKey**：您可以在 Gate.io 账户中申请新的 API 密钥，只需将值复制到此属性即可。如果设置了 APIKey，客户端将对私有服务器进行身份验证。如果为空，则只有公共频道可用。
- **ApiSecret**：API 的密钥值。

连接

当客户端成功连接到 Gate.io 服务器时，触发 **OnConnect** 事件。触发 **OnConnect** 事件后，您可以开始向 Gate.io 服务器发送和接收消息。如果您正在连接私有 WebSocket 频道，则必须等到触发 **OnGateIOAuthentication** 事件且检查 success 参数为 True 后，才能订阅任何私有频道。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWS_API_GateIO *oGateIO = new TsgcWS_API_GateIO();
oGateIO->Client = oClient;
oGateIO->GateIO->ApiKey = "your_api_key";
oGateIO->GateIO->ApiSecret = "your_api_secret";
oGateIO->GateIOChannel = giocSpot;
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
    DoLog("#GateIO Connected");
}
```

事件

GateIO 客户端实现了以下事件，用于控制连接流程并获取 WebSocket 服务器发送的数据：

- **OnConnect**：当 WebSocket 客户端连接到 WebSocket 服务器时触发。
- **OnDisconnect**：当 WebSocket 客户端从 WebSocket 服务器断开连接时触发。
- **OnGateIOAuthentication**：当客户端针对私有 WebSocket 服务器进行身份验证时触发。
- **OnGateIOSubscribe**：当客户端订阅 WebSocket 通道时触发。
- **OnGateIOUnSubscribe**：当客户端取消订阅 WebSocket 频道时。
- **OnGateIOData**：当客户端从服务器接收数据时触发。
- **OnGateIOError**：当 GateIO WebSocket 连接发生任何错误时。
- **OnGateIOHTTPException**：当 REST 请求期间出现任何错误时触发。

WebSocket API

WebSocket 数据流提供订单和交易的实时市场数据更新。

您可以订阅以下频道：

方法	公共或私有	描述
SubscribeTicker	公开	订阅某交易对的行情流。
SubscribeTrades	公开	订阅最新成交流。

SubscribeCandlesticks	公开	订阅 K 线数据流。支持多个时间间隔（如 1m、5m、15m、30m、1h）。
SubscribeOrderBook	公开	订阅订单簿流。支持可配置的深度级别和更新间隔。
SubscribeOrders	私有	订阅订单更新。需要身份验证。
SubscribeBalances	私有	订阅余额更新。需要认证。

以下是订阅公共 WebSocket 频道的示例。

```
void OnConnect(TsgcWSConnection *Connection)
{
    oGateIO->SubscribeTicker("BTC_USDT");
    oGateIO->SubscribeOrderBook("BTC_USDT", "20", "100ms");
}
```

REST API

REST API 提供公开和私有方法，用于从市场和私人账户请求数据。通过组件的 **REST_API** 属性访问 REST API。

方法	公开 / 私有	描述
GetCurrencyPairs	公开	获取可用货币对的信息。
GetTickers	公开	获取一个或所有货币对的行情信息。
GetOrderBook	公开	获取货币对的订单簿。
GetCandlesticks	公开	获取货币对的蜡烛图/K 线数据。
GetTrades	公开	获取某货币对的最新成交记录。
PlaceOrder	私有	下新订单。支持限价单和市价单。
CancelOrder	私有	取消现有订单。
GetOrder	私有	获取特定订单的详情。
GetOpenOrders	私有	获取未结订单列表。
GetSpotAccounts	私有	获取现货账户信息。

以下是使用 REST API 的示例。

```
TsgcWS_API_GateIO *oGateIO = new TsgcWS_API_GateIO();
oGateIO->GateIO->ApiKey = "your_api_key";
oGateIO->GateIO->ApiSecret = "your_api_secret";
// Get tickers
oGateIO->REST_API->GetTickers("BTC_USDT");
// Place an order
oGateIO->REST_API->PlaceOrder("BTC_USDT", "buy", "0.001", "50000", "limit");
```

API Deribit

Deribit

Deribit 是一家加密货币衍生品交易所，提供比特币和以太坊的期货和期权交易。

支持的 API

- **WebSocket API**：连接到 WebSocket 服务器，提供实时市场数据更新、账户变动等信息。
- **REST API**：发送 HTTP 请求以获取市场数据、下单、账户数据等。

属性

您可以在 Deribit 属性中配置以下属性。

- **ApiKey**：您可以在 Deribit 账户中申请新的 API 密钥，只需将值复制到此属性。如果设置了 ApiKey，客户端将对私有服务器进行身份验证。如果为空，则只有公共频道可用。
- **ApiSecret**：API 的密钥值。
- **TestNet**：若启用，将连接到 Deribit TestNet 环境（默认禁用）。适用于无需使用真实资金的测试。

连接

当客户端成功连接到 Deribit 服务器时，将触发事件 **OnConnect**。触发 **OnConnect** 事件后，您可以开始向 Deribit 服务器发送和接收消息。如果您正在连接私有 WebSocket 通道，则必须等待 **OnDeribitAuthentication** 事件触发，并确认 success 参数为 true，然后再订阅任何私有通道。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWS_API_Deribit *oDeribit = new TsgcWS_API_Deribit();
oDeribit->Client = oClient;
oDeribit->Deribit->ApiKey = "your_api_key";
oDeribit->Deribit->ApiSecret = "your_api_secret";
oDeribit->Deribit->TestNet = false;
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
    DoLog("#Deribit Connected");
}
```

事件

Deribit 客户端实现了以下事件，用于控制连接流程并获取 WebSocket 服务器发送的数据：

- **OnConnect**：当 WebSocket 客户端连接到 WebSocket 服务器时触发。
- **OnDisconnect**：当 WebSocket 客户端从 WebSocket 服务器断开连接时触发。
- **OnDeribitAuthentication**：当客户端向私有 WebSocket 服务器进行身份验证时触发。
- **OnDeribitSubscribe**：当客户端订阅 WebSocket 频道时触发。
- **OnDeribitUnSubscribe**：当客户端取消订阅 WebSocket 频道时触发。
- **OnDeribitData**：当客户端从服务器接收到数据时。
- **OnDeribitError**：Deribit WebSocket 连接期间发生任何错误时触发。
- **OnDeribitHTTPException**：REST 请求期间发生任何错误时触发。

WebSocket API

WebSocket 数据流提供订单和交易的实时市场数据更新。

您可以订阅以下公共频道：

方法	描述
SubscribeTicker	订阅某个合约的行情流。提供实时价格和交易量数据。
SubscribeTrades	订阅某一品种的最新成交流。
SubscribeOrderBook	订阅订单簿数据流。支持可配置的分组、深度级别和更新间隔。

SubscribeCandle	订阅 K 线流。支持多种时间分辨率（例如 1、3、5、10、15、30、60、120、180、360、720、1D）。
SubscribePerpetual	订阅某个合约的永续合约数据，提供资金费率和其他永续合约特定信息。
SubscribeBookChange	订阅某合约的订单簿变动通知。

您可以订阅以下私有频道（需要认证）：

方法	描述
SubscribeOrders	订阅某一标的的订单更新。
SubscribePositions	订阅持仓更新。支持按货币和合约种类过滤。
SubscribeAccountChanges	订阅某种货币的账户余额变动。
SubscribeUserTrades	订阅某个合约的用户成交更新。

以下是订阅 WebSocket 频道的示例。

```
void OnDeribitAuthentication(TObject *Sender, bool aSuccess, const string aError, const string aRawMessage)
{
    if (aSuccess == true)
    {
        oDeribit->SubscribeOrders("BTC-PERPETUAL");
        oDeribit->SubscribePositions("BTC");
    }
}
```

REST API

REST API 提供公共和私有方法，用于从市场、交易、账户和钱包请求数据。通过组件的 **REST_API** 属性访问 REST API。

市场数据（公开）

方法	描述
GetInstruments	获取某种货币的可用工具。
GetTicker	获取某一金融工具的行情信息。
GetOrderBook	获取某个金融工具的订单簿。
GetTrades	获取某个合约的最新成交记录。
GetCurrencies	获取支持的货币列表。
GetIndexPrice	获取指定索引名称的当前指数价格。
GetFundingRateHistory	获取某合约的资金费率历史。
GetFundingRateValue	获取某一合约的当前资金费率值。
GetBookSummaryByCurrency	获取某种货币所有合约的账面摘要。
GetBookSummaryByInstrument	获取特定合约的账本摘要。

交易（私有）

方法	描述
Buy	下买入单。支持市价和限价订单类型。
卖出	下卖单。支持市价单和限价单类型。
CancelOrder	通过订单 ID 取消现有订单。
CancelAllOrders	取消所有未成交订单。
CancelAllByInstrument	取消特定合约的所有未结挂单。
EditOrder	编辑现有订单（修改数量或价格）。
GetOpenOrders	获取某个货币的未结订单列表。
GetOrderState	获取特定订单的状态。
GetOrderHistory	获取某货币的订单历史记录。

账户（私有）

方法	描述
GetPositions	获取某一货币的持仓信息。
GetAccountSummary	获取某货币的账户摘要。
GetSubAccounts	获取子账户列表。
GetTransactionLog	获取某种货币的交易日志。

钱包（私有）

方法	描述
GetDeposits	获取某一货币的充值历史。
GetWithdrawals	获取某种货币的提币历史记录。
GetTransfers	获取某种货币的转账历史记录。

以下是使用 REST API 的示例。

```
TsgcWS_API_Deribit *oDeribit = new TsgcWS_API_Deribit();
oDeribit->Deribit->ApiKey = "your_api_key";
oDeribit->Deribit->ApiSecret = "your_api_secret";
// Get ticker
oDeribit->REST_API->GetTicker("BTC-PERPETUAL");
// Place a buy order
oDeribit->REST_API->Buy("BTC-PERPETUAL", 10, "market");
// Get account summary
oDeribit->REST_API->GetAccountSummary("BTC");
```


API Crypto.com

[Crypto.com](#)

支持的 API

- **WebSocket API**：连接到 WebSocket 服务器，提供实时市场数据更新、账户变动等信息。
- **REST API**：发送 HTTP 请求以获取市场数据、下单、账户数据等。

属性

使用 **Channel** 属性选择频道类型：

- **cccMarket**：连接到用于公开市场数据的市场 WebSocket API。
- **cccUser**：连接到用于私有账户数据的用户 WebSocket API（需要身份验证）。

您可以在 CryptoCom 属性中配置以下属性。

- **ApiKey**：您可以在 Crypto.com 账户中申请新的 API 密钥，只需将其值复制到此属性即可。私有通道和需要身份验证的 REST API 调用必须提供此密钥。
- **ApiSecret**：API 的密钥值。

连接

当客户端成功连接到 Crypto.com 服务器时，将触发事件 **OnConnect**。触发 **OnConnect** 事件后，您可以开始向/从 Crypto.com 服务器发送和接收消息。如果您要连接到用户频道，必须等到触发 **OnCryptoComAuthentication** 事件并检查 success 参数是否为 true，然后才能订阅任何私有频道。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWS_API_CryptoCom *oCryptoCom = new TsgcWS_API_CryptoCom();
oCryptoCom->Client = oClient;
oCryptoCom->CryptoCom->ApiKey = "your_api_key";
oCryptoCom->CryptoCom->ApiSecret = "your_api_secret";
oCryptoCom->Channel = cccMarket;
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
    DoLog("#CryptoCom Connected");
}
```

事件

Crypto.com 客户端实现了以下事件，用于控制连接流程并获取 WebSocket 服务器发送的数据：

- **OnConnect**：当 WebSocket 客户端连接到 WebSocket 服务器时触发。
- **OnDisconnect**：当 WebSocket 客户端从 WebSocket 服务器断开连接时触发。
- **OnCryptoComAuthentication**：当客户端向私有 WebSocket 服务器进行认证时触发。
- **OnCryptoComSubscribe**：当客户端订阅 WebSocket 频道时触发。
- **OnCryptoComUnSubscribe**：当客户端取消订阅 WebSocket 频道时触发。
- **OnCryptoComData**：当客户端从服务器接收到数据时触发。
- **OnCryptoComError**：当 Crypto.com WebSocket 连接发生任何错误时。
- **OnCryptoComHTTPException**：REST 请求期间发生任何错误时触发。

WebSocket API

WebSocket 数据流提供订单和交易的实时市场数据更新。

您可以订阅以下**市场频道**（公开）：

方法	描述
SubscribeTicker	订阅某个合约的行情流。提供实时价格和交易量数据。
SubscribeTrade	订阅某一品种的最新成交流。
SubscribeCandlestick	订阅 K 线数据流，支持多个时间间隔（如 5m、15m、30m、1h、4h、1D）。

SubscribeBook	订阅订单簿数据流，支持可配置的深度级别（如 10、50）。

您可以订阅以下**用户频道**（私有，需要身份验证）：

方法	描述
SubscribeOrders	订阅订单更新，可选择按合约名称过滤。
SubscribeUserTrades	订阅用户交易更新。可选按合约名称过滤。
SubscribeBalance	订阅您账户的余额更新。

以下是订阅市场频道的示例。

```
void OnConnect(TsgcWSConnection *Connection)
{
    oCryptoCom->SubscribeTicker("BTC_USDT");
    oCryptoCom->SubscribeBook("BTC_USDT", "10");
}
```

REST API

REST API 提供公开和私有方法，用于从市场和私人账户请求数据。通过组件的 **REST_API** 属性访问 REST API。

方法	公开 / 私有	描述
GetInstruments	公开	获取可用合约列表。
GetTicker	公开	获取一个或所有合约的行情信息。
GetOrderBook	公开	获取某个金融工具的订单簿。
GetTrades	公开	获取某个合约的最新成交记录。
GetCandlestick	公开	获取某个合约的K线/蜡烛图数据。
CreateOrder	私有	创建新订单。支持限价单和市价单。
CancelOrder	私有	取消现有订单。
CancelAllOrders	私有	撤销某个合约的所有未平仓订单。
GetOpenOrders	私有	获取未结订单列表。
GetOrderDetail	私有	获取特定订单的详情。
GetOrderHistory	私有	获取历史订单。
GetAccountSummary	私有	获取账户摘要信息。

以下是使用 REST API 的示例。

```
TsgcWS_API_CryptoCom *oCryptoCom = new TsgcWS_API_CryptoCom();
oCryptoCom->CryptoCom->ApiKey = "your_api_key";
oCryptoCom->CryptoCom->ApiSecret = "your_api_secret";
// Get instruments
oCryptoCom->REST_API->GetInstruments();
// Create an order
oCryptoCom->REST_API->CreateOrder("BTC_USDT", "BUY", "MARKET", "", "0.001");
// Get account summary
oCryptoCom->REST_API->GetAccountSummary();
```

API HTX

HTX

HTX（前身为 Huobi）是一家国际多语言加密货币交易所。HTX API 组件提供 HTTP REST API，以补充现有的 Huobi WebSocket API。

支持的 API

- **WebSocket API**：连接到 WebSocket 服务器，提供实时市场数据更新、账户变动等。有关 WebSocket 订阅方法，请参阅 Huobi API 文档。
- **REST API**：发送 HTTP 请求以获取市场数据、下单、账户数据等。

属性

您可以在 Huobi 属性（用于 WebSocket）或 HTXOptions 属性（用于 REST API）中配置以下属性。

- **ApiKey**：您可以在 HTX 账户中申请新的 API 密钥，将值复制到此属性。如果设置了 ApiKey，客户端将连接到 WebSocket 私有服务器；如果为空，则连接到 WebSocket 公共服务器。
- **ApiSecret**：API 的密钥值。

连接

HTX 组件扩展了 Huobi WebSocket API 客户端。当客户端成功连接到 HTX 服务器时，**OnConnect** 事件触发。如果 ApiKey 不为空，客户端将尝试连接到私有 WebSocket 服务器，届时只有私有方法可用。如果 ApiKey 为空，客户端将连接到公共 WebSocket 服务器，只有公共方法可用。

```
TsgcWebSocketClient *oClient = new TsgcWebSocketClient();
TsgcWS_API_Huobi *oHTX = new TsgcWS_API_Huobi();
oHTX->Client = oClient;
oHTX->Huobi->ApiKey = "your_api_key";
oHTX->Huobi->ApiSecret = "your_api_secret";
oClient->Active = true;
void OnConnect(TsgcWSConnection *Connection)
{
    DoLog("#HTX Connected");
}
```

事件

HTX 客户端实现了以下事件，用于控制连接流程并获取 WebSocket 服务器发送的数据：

- **OnConnect**：当 WebSocket 客户端连接到 WebSocket 服务器时触发。
- **OnDisconnect**：当 WebSocket 客户端从 WebSocket 服务器断开连接时触发。
- **OnHuobiSubscribed**：成功订阅后调用的事件。
- **OnHuobiUnSubscribed**：成功取消订阅后调用的事件。
- **OnHuobiUpdate**：每当数据有更新时（K 线、市场深度.....）调用此事件。
- **OnHuobiError**：如果 HTX API 出现错误，此事件将提供有关该错误的信息。

WebSocket API

WebSocket API 提供实时市场数据更新。有关 WebSocket 订阅方法的完整列表，请参阅 Huobi API 文档，包括公共频道（SubscribeKLine、SubscribeMarketDepth、SubscribeTradeDetail 等）和私有频道（SubscribeOrderUpdates、SubscribeTradeClearing、SubscribeAccountChange）。

REST API

REST API 提供公开和私有方法，用于从市场和私人账户请求数据。REST API 通过 **TsgcHTTP_API_HTX** 组件访问。

市场数据（公开）

方法	描述
GetServerTime	获取服务器时间。
GetSymbols	获取可用交易标的列表。
GetCurrencies	获取支持的货币列表。
GetMarketTickers	获取所有交易对的行情。
GetMarketDetail	获取某一交易对的 24 小时市场详情。
GetMarketDepth	获取某一交易对的订单簿深度。支持可配置的深度类型和深度档位。
GetMarketTrade	获取某个符号的最新成交。
GetMarketHistoryTrade	获取某个交易对的最新成交历史记录。
GetMarketHistoryKline	获取 K 线数据。支持多种周期（1min、5min、15min、30min、60min、4hour、1day、1mon、1week、1year）。

交易（私有）

方法	描述
PlaceOrder	下新订单。支持市价买入、市价卖出、限价买入、限价卖出及其他订单类型。
CancelOrder	通过订单 ID 取消现有订单。
GetOrder	获取特定订单的详情。
GetOpenOrders	获取账户的未结订单列表。
GetOrderHistory	获取某个交易对的订单历史。

账户（私有）

方法	描述
GetAccounts	获取账户列表。
GetAccountBalance	获取特定账户的余额。
GetAccountAssetValuation	获取某种账户类型（例如现货）的总资产估值。

以下是使用 REST API 的示例。

```
TsgcHTTP_API_HTX *oHTX = new TsgcHTTP_API_HTX();
oHTX->HTXOptions->ApiKey = "your_api_key";
oHTX->HTXOptions->ApiSecret = "your_api_secret";
// Get market tickers
oHTX->GetMarketTickers();
// Get kline data
oHTX->GetMarketHistoryKline("btcusdt", "1min", 150);
// Place an order
oHTX->PlaceOrder("12345678", "btcusdt", "buy-limit", "0.001", "50000");
// Get account balance
oHTX->GetAccountBalance("12345678");
```

WhatsApp Cloud API

Whatsapp

发送和接收消息，使用云托管版本的 **WhatsApp Business 平台**。**Cloud API** 让您无需自行托管服务器即可实现 WhatsApp Business API，同时更轻松地扩展您的商业消息能力。Cloud API 支持每秒最多 80 条消息的综合收发（包括文本和媒体消息）。

WhatsApp Business API 允许中大型企业与其客户进行大规模沟通。通过该 API，企业可以构建将数千名客户与客服代表或机器人相连的系统，支持程序化和手动两种通信方式。此外，您还可以将该 API 与 CRM 和营销平台等众多后端系统集成。

功能特性

企业将通过 Cloud API 更快获得所有新功能。目前，WhatsApp Business Cloud API 包含 WhatsApp Business API 所有可用功能。

WhatsApp Cloud API 的实用功能：

- **集成**WhatsApp 消息与 **CRM**、**分析**和**第三方应用**
- **绿色勾选**，已验证的 WhatsApp Business 资料
- WhatsApp **广播 & 批量消息**
- 无应用程序或界面，通过 BSP 或 CRM 使用
- **WhatsApp Chatbot** 及使用第三方应用的聊天**自动化**
- 大规模**定时发送** WhatsApp 消息。
- **互动消息**功能包括列表消息、回复按钮、CTA 消息

最常见的用途

- **配置**
 - [WhatsApp 创建应用程序](#)
 - [WhatsApp 电话号码 ID](#)
 - [WhatsApp 令牌](#)
 - [WhatsApp Webhook](#)
 - [WhatsApp 安全性](#)
- **消息**
 - [WhatsApp 发送消息](#)
 - [WhatsApp 发送交互消息](#)
 - [WhatsApp 发送模板消息](#)
 - [WhatsApp 接收消息和状态通知](#)
 - [WhatsApp 发送文件](#)
 - [WhatsApp 下载媒体](#)

开始使用

要使用测试号码发送和接收第一条消息，请完成以下步骤：

1. 设置开发者资产和平台访问权限

- [注册为 Meta 开发者](#)
- [为您的账户启用双因素身份验证](#)
- **创建 Meta 应用**：访问 developers.facebook.com > 我的应用 > 创建应用。选择"Business"类型并按照屏幕上的提示操作。

在应用仪表板中，点击您希望连接到 WhatsApp 的应用。向下滚动找到"WhatsApp"产品，然后点击 **设置**。

接下来，您将看到选择现有 Business Manager 的选项（如果您有的话），或者，如果您愿意，入驻流程也可以自动为您创建一个（您可以在之后对业务进行自定义）。做出选择后，点击 **继续**。

当您点击 **继续**时，引导流程将执行以下操作：

- 您的应用已与您选择或自动创建的 Business Manager 关联。
- 您的企业已添加一个 WhatsApp 测试电话号码。您可以使用此测试电话号码探索 WhatsApp Business 平台，无需注册或迁移真实电话号码。测试电话号码最多可向 5 位收件人（可在全球任何地方）发送无限条消息。

2. 发送测试消息

现在，您可以打开 IDE 并创建新项目。放置一个 TsgcWhatsapp_Client 组件并填写以下属性：

- **WhatsappOptions.PhoneNumberId**：是用于发送消息的电话号码 ID。
- **WhatsappOptions.Token**：有效期 24 小时的临时访问令牌。

正确配置这 2 个属性后，调用 **SendTest** 方法，通过 **WhatsApp Business Platform** 向电话号码发送您的**第一条消息**。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendTest("34605889421");
```

3. 配置 Webhook

要在收到消息或消息状态发生变化时收到提醒，您需要为您的应用程序设置 Webhooks 端点。设置 Webhooks 不会影响您的电话号码状态，也不会干扰您发送或接收消息。

若要开始，首先需要创建端点，请先配置 WhatsApp 客户端组件的 **ServerOptions** 属性，并配置以下属性：

- **ServerOptions**：在此处您可以配置要绑定的 IP 地址、监听端口，以及是否使用 SSL（WebHook 必须在安全服务器上运行，您可以将服务器配置为使用 SSL 或将 WebHook 请求代理到非 HTTPS 服务器）。服务器基于 [TsgcWebSocketHTTPServer](#)。
 - **WebhookOptions**：此属性允许您设置 Webhook 属性，稍后将在您的 Facebook 开发者账户中配置。
 - **Endpoint**：端点名称，默认为 /webhook。示例：如果您的服务器监听 https://www.esegece.com，则端点为 "https://www.esegece.com/webhook"
 - **Token**：安全字符串，可包含您定义的任意值。用于验证 Webhook 注册是否正确。

配置服务器后，您可以使用方法 **StartServer** 启动服务器并接受传入请求。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->ServerOptions->WebhookOptions->PhoneNumberId = "/webhook";
oClient->ServerOptions->WebhookOptions->Token = "MySecretToken";
oClient->StartServer();
```

端点准备好后，请前往您的应用控制台。

在您的应用程序仪表板中，找到 WhatsApp 产品并单击 **Configuration**。然后，找到 Webhooks 部分并单击 **Configure a webhook**。单击后，屏幕上将出现一个对话框，要求您提供两项内容：

- 回调 URL：这是 Meta 将向其发送事件的 URL。
- 验证令牌：此字符串由您在创建 Webhook 端点时自行设置。

添加信息后，点击 **验证并保存**。

回到应用程序仪表板，在左侧面板中点击 **WhatsApp > Configuration** 在左侧面板中。在 Webhooks 下，点击 **Manage**。将打开一个对话框，显示您可以收到通知的所有对象。要接收用户的消息，请为 **messages** 点击 **Subscribe**。

4. 接收测试消息

每次收到新消息时，都会调用客户端事件 **OnMessageReceived**。

```
void OnMessageReceived(TObject *Sender, TsgcWhatsapp_Receive_Message *aMessage, ref bool aMarkAsRead)
{
    DoLog("Received: " + aMessage->Messages->_Message[0]->Id);
}
```

现在您的 Webhook 已设置完毕，向您使用的测试号码发送一条消息。您应该立即收到包含消息内容的 Webhook 通知！

WhatsApp API 不允许向从未联系过您（最近 24 小时内）的手机发送自由文本消息。向从未向您的开发者账号发送过 WhatsApp 消息的手机发送文本消息的唯一方式，是发送模板消息（须经 Meta 预先批准）。若要在测试中绕过此限制以发送自由文本消息，请先从目标手机向您的开发者账号发送一条 WhatsApp 消息，之后即可在 24 小时内发送自由文本消息。

事件

OnBeforeSendMessage

此事件在消息发送到 WhatsApp 服务器之前调用。您可以通过 RawMessage 参数访问内部消息。

OnBeforeSubscribe

此事件在服务器订阅主题之前调用。使用 Accept 参数允许或拒绝订阅。默认情况下，服务器将订阅所有请求的事件。

OnRawMessage

当服务器收到尚未解析的新消息时调用此事件，因此您可以访问原始消息。

OnMessageReceived

此事件在服务器接收并解析新消息后调用。如果将 MarkAsRead 参数设置为 True，发送方将收到双勾确认。

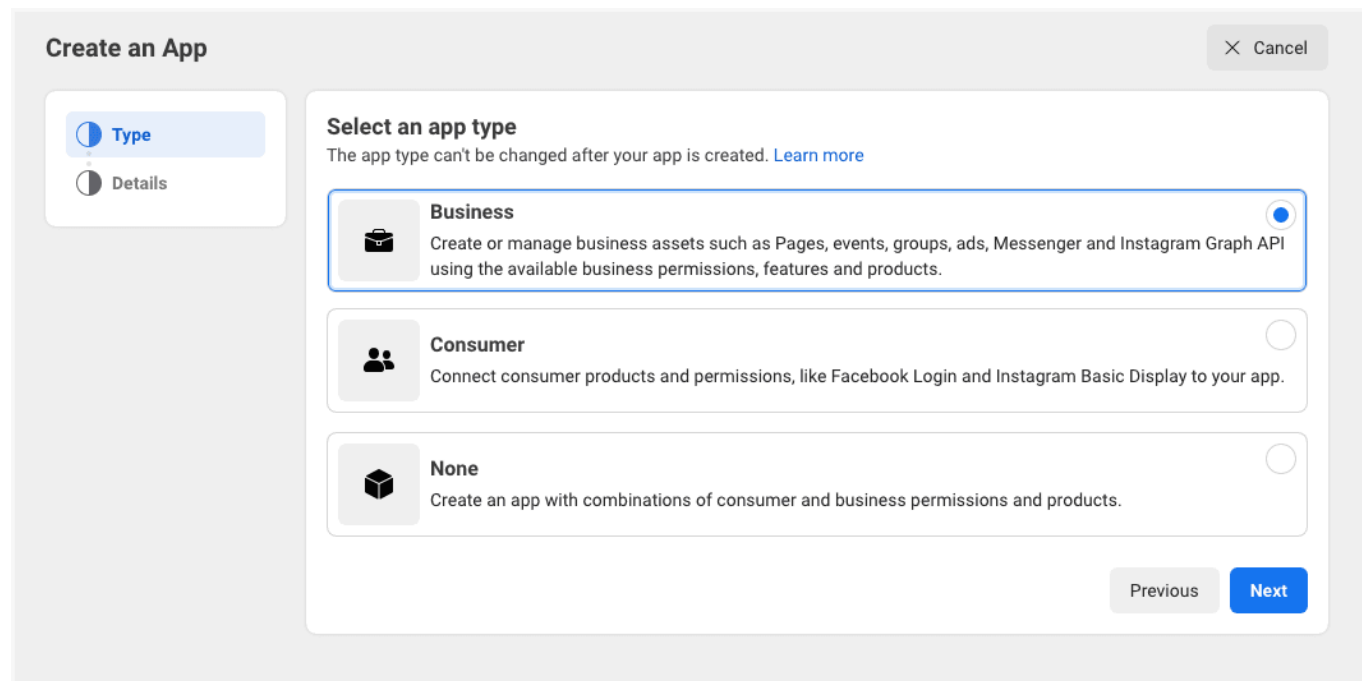
OnMessageSent

每次服务器收到关于之前已发送消息的新状态消息时，都会调用此事件。读取 Status 属性可了解消息是否已发送、已送达或已读。

WhatsApp 创建应用程序

前往 developers.facebook.com 并创建应用。

选择 **Business Type** 作为应用类型并继续。



Create an App ✕ Cancel

Type Details

Select an app type
The app type can't be changed after your app is created. [Learn more](#)

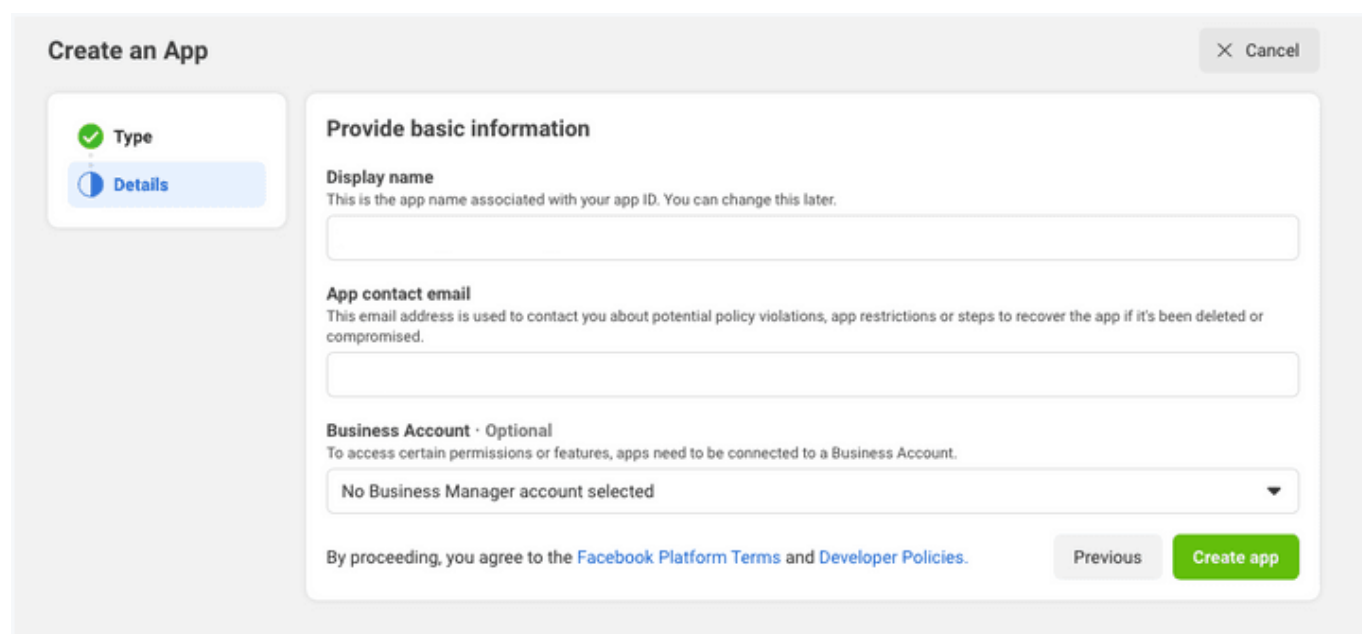
Business ☒
Create or manage business assets such as Pages, events, groups, ads, Messenger and Instagram Graph API using the available business permissions, features and products.

Consumer ☐
Connect consumer products and permissions, like Facebook Login and Instagram Basic Display to your app.

None ☐
Create an app with combinations of consumer and business permissions and products.

Previous Next

为您的应用提供一个名称（请避免使用"WhatsApp"或"Facebook"等商标名称）。



Create an App ✕ Cancel

Type Details

Provide basic information

Display name
This is the app name associated with your app ID. You can change this later.

App contact email
This email address is used to contact you about potential policy violations, app restrictions or steps to recover the app if it's been deleted or compromised.

Business Account · Optional
To access certain permissions or features, apps need to be connected to a Business Account.

No Business Manager account selected

By proceeding, you agree to the [Facebook Platform Terms](#) and [Developer Policies](#).

Previous Create app

应用创建后，在下一个屏幕上点击 **WhatsApp 按钮**，为您的应用添加 WhatsApp 发送功能。



在下一屏中，您需要将 WhatsApp 应用与您的 Facebook 企业账户关联。如果您还没有企业账户，也可以选择在此创建一个新的企业账户。

WhatsApp 手机号码 ID

注册 WhatsApp Cloud API 后，Facebook 将提供一个测试用 WhatsApp 电话号码，作为应用程序的默认发送地址。在开发阶段，收件人最多可添加 5 个电话号码，无需进行任何付款。

之后，您可以注册自己的手机号码，以解除 5 个手机号码的限制。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();  
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
```

WhatsApp Token

WhatsApp Cloud API 需要有效的令牌才能使用 Cloud API 发送任何消息。

Facebook 提供了一个测试 WhatsApp 电话号码，允许您向最多 5 个电话号码发送消息。您可以通过注册自己的电话号码来解除此限制。

WhatsApp API 提供一个**临时访问令牌**，有效期 23 小时。必须在 TsgcWhatsApp_Client 组件中配置此令牌才能发送消息。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();  
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
```

如果您需要长期有效的令牌，可以创建（或更新）一个系统用户，并使用 **whatsapp_business_messaging** 权限生成新令牌。这将允许您发送和接收 WhatsApp 消息，而无需每 23 小时更新一次令牌。

WhatsApp Webhook

订阅 Webhook，以接收有关您的企业收到的消息和客户资料更新的通知。

创建端点

在开始接收通知之前，您需要在服务器上创建一个端点来接收通知。

您的端点必须能够处理两种类型的 HTTPS 请求：验证请求和事件通知。由于两种请求都使用 HTTPS，您的服务器必须正确配置并安装有效的 TLS 或 SSL 证书。不支持自签名证书。

在 WhatsApp 设置中配置 Webhook 时，必须定义服务器监听的端点以及一个可以是任意值的令牌，此令牌在注册 Webhook 端点时使用，用于验证订阅者的有效性。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->ServerOptions->WebhookOptions->PhoneNumberId = "/webhook";
oClient->ServerOptions->WebhookOptions->Token = "MySecretToken";
oClient->StartServer();
```

配置 Webhook 后，订阅消息 Webhook 字段，以便在每次收到新消息时收到通知。

您可以阅读有关配置 [SSL 服务器](#) 的更多信息。

WhatsApp 安全

每次收到新消息或消息状态有更新时，服务器都会在[Webhook](#) 中配置的端点收到通知。为确保请求来自 WhatsApp Cloud API 服务器，请求标头中包含签名，您可以配置 WhatsApp 客户端在处理消息之前验证签名。

为此，首先需要在属性 **ServerOptions.Application.Secret** 中设置 Application Secret，并启用 **VerifySignature** 属性。

配置完成后，每次收到新消息时，系统首先验证签名；若签名错误，则返回 500 错误，消息不予处理。

WhatsApp 发送消息

所有 API 调用必须使用访问令牌进行认证。开发者可以使用在[应用仪表盘 > WhatsApp > 入门指南](#)中生成的访问令牌认证 API 调用

API 调用以字符串形式返回消息 ID。

文本消息

调用方法 **SendMessageText** 并传入以下参数：

- **aTo**：电话号码
- **aText**：消息的文本。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageText("34605889421", "Hello from sgcWebSockets!!!");
```

图像消息

调用方法 **SendMessageImage** 并传入以下参数：

- **aTo**：电话号码
- **aLink**：要发送的图像的 URL
- **aCaption**：图片标题（可选）。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageImage("34605889421", "https://www.media.com/image.png", "logo");
```

文档消息

调用 **SendMessageDocument** 方法并传入以下参数：

- **aTo**：电话号码
- **aLink**：待发送文档所在的 URL
- **aCaption**：文档标题（可选）。
- **aFileName**：文件名（可选）。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageDocument("34605889421", "https://www.documents.com/file.txt", "Document", "file.txt");
```

音频消息

调用方法 **SendMessageAudio** 并传入以下参数：

- **aTo**：电话号码
- **aLink**：要发送的音频所在的 URL

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageAudio("34605889421", "https://www.audio.com/audio.mp3");
```

视频消息

调用方法 **SendMessageVideo** 并传递以下参数：

- **aTo**：电话号码
- **aLink**：要发送的视频的 URL

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageVideo("34605889421", "https://www.video.com/audio.mp4");
```

贴纸消息

调用方法 **SendMessageSticker**，并传入以下参数：

- **aTo**：电话号码
- **aLink**：要发送的贴纸所在的 URL

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageSticker("34605889421", "https://www.stickers.com/sticker");
```

位置消息

调用 **SendMessageLocation** 方法，并传入以下参数：

- **aTo**：电话号码
- **aLongitude**：位置的经度。
- **aLatitude**：位置的纬度。
- **aName**：位置名称。
- **aAddress**：位置地址。仅在设置了 **aName** 时显示。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageLocation("34605889421", "50.159305", "9.762686", "My Location", "My Address");
```

联系消息

调用方法 **SendMessageContact** 并传递以下参数：

- **aTo**：电话号码
- **aName**：全名，通常显示的形式（必填）。
- **aPhone**：电话号码（可选）。
- **aEmail**：电子邮件地址（可选）。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageLocation("34605889421", "John Smith", "15550386570", "john@mail.com");
```

WhatsApp 发送交互式消息

交互式消息为您的用户提供了一种更简便的方式，让他们能在 WhatsApp 上找到并选择您企业提供的內容。在测试中，使用交互式消息功能的聊天机器人与纯文本聊天机器人相比，实现了显著更高的响应率和转化率。

以下消息被视为交互式消息：

- **列表消息**：包含最多 10 个选项菜单的消息。这种消息类型为用户在与企业互动时提供了更简单、更一致的选择方式。
- **回复按钮**：包含最多 3 个选项的消息，每个选项是一个按钮。这种消息类型为用户在与企业互动时从菜单中进行选择提供了更快捷的方式。

交互式消息规范

- 交互式消息可以在同一流程中组合使用。
- 用户无法同时从列表或按钮消息中选择多个选项，但可以返回并重新打开之前的消息。
- 列表或回复按钮消息不能用作通知。目前，它们只能在用户发送最后一条消息后的 24 小时内发送。如果您尝试在 24 小时窗口之外发送消息，将收到错误消息。

何时应使用

列表消息最适合呈现多个选项，例如：

- 客户服务或常见问题菜单
- 外卖菜单
- 附近商店或地点的选择
- 可用预订时间
- 选择最近的订单重复

回复按钮最适合提供来自有限选项集的快速响应，例如：

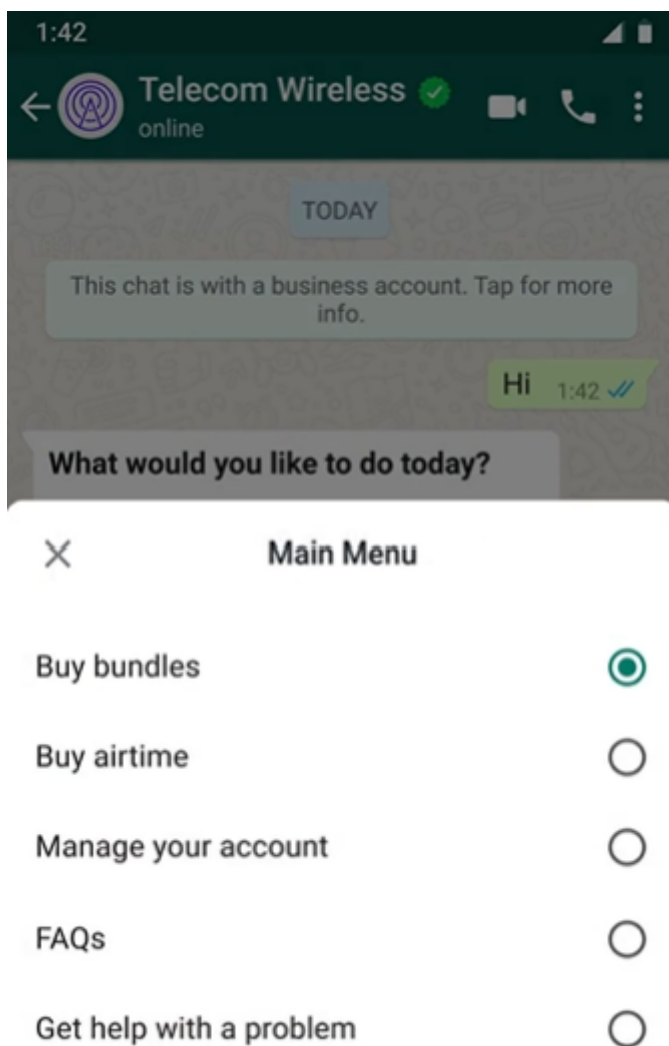
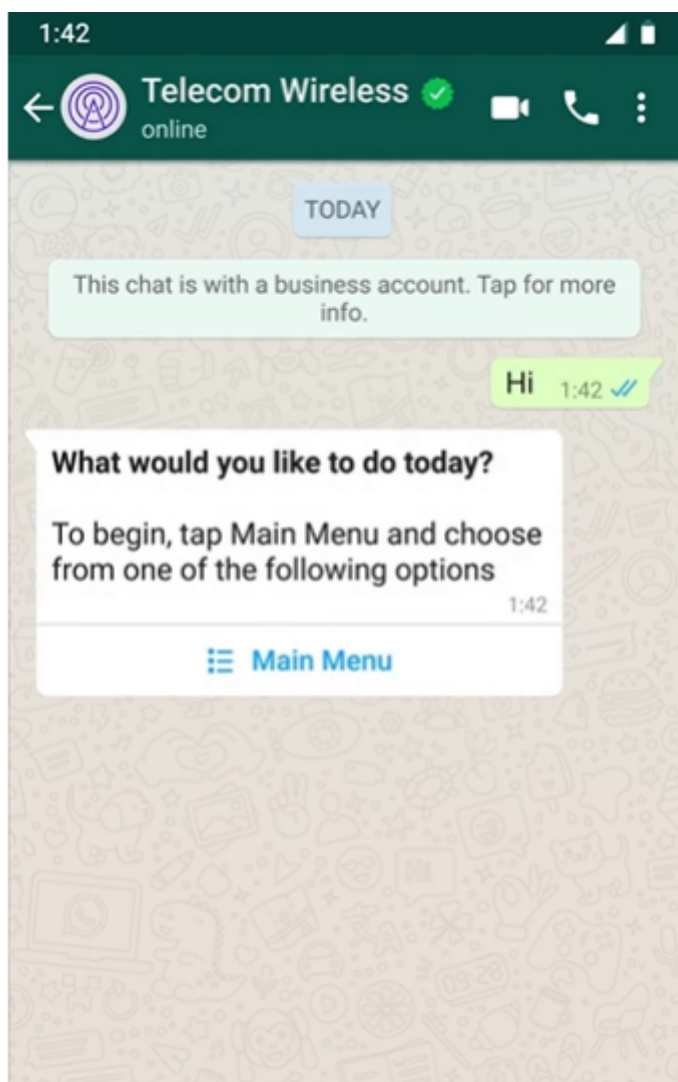
- 话费充值
- 修改个人信息
- 重新排序之前的订单
- 请求退货
- 向食物订单添加可选附加项
- 选择支付方式

回复按钮对于“个性化”用例特别有价值，在这些场景中通用响应并不适用。

交互式列表

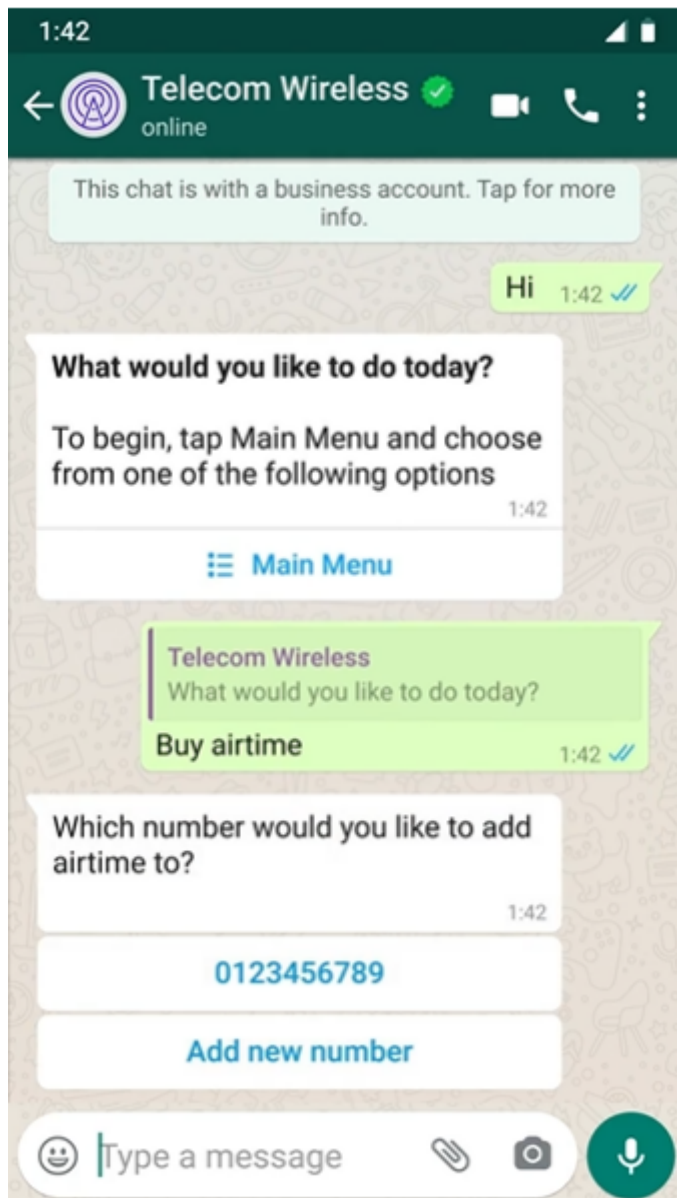
```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageInteractiveList("34605889421
```

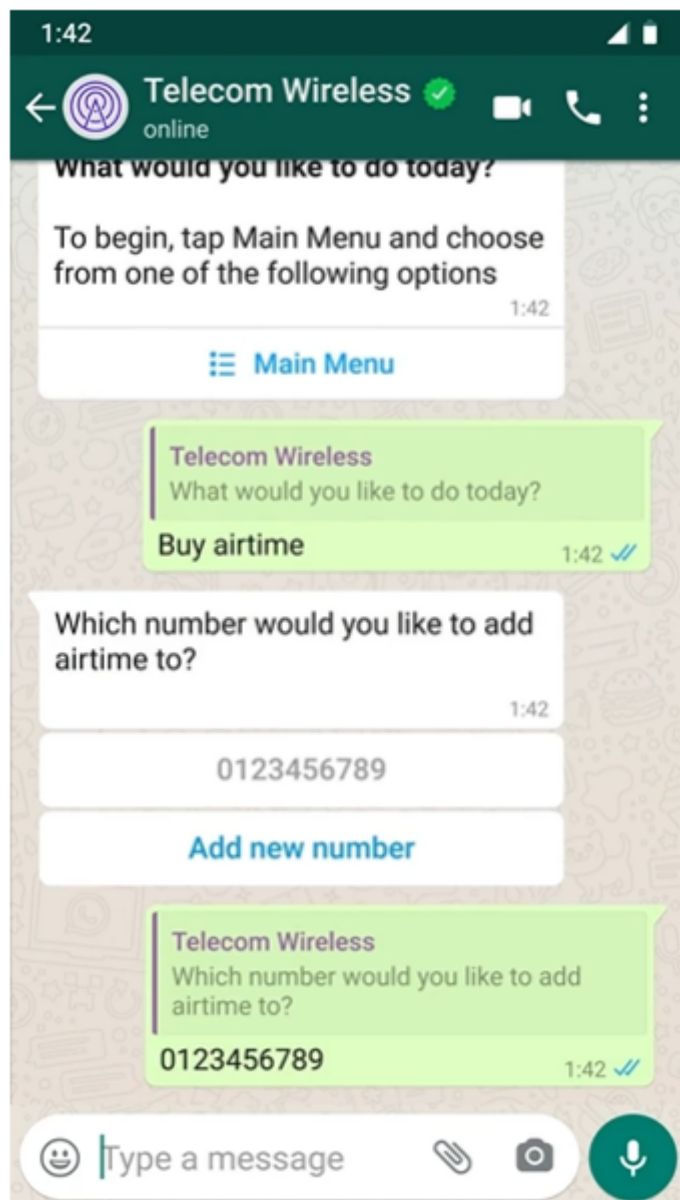
```
", "What Would you like to do today?", "To begin, Tap Main Menu and choose from of the following options", "", "M
```



回复按钮

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageInteractiveButtons("34605889421", "Select an option", "Which number would you like to add air
```





WhatsApp 发送模板消息

调用方法 `SendMessageTemplate` 并传入以下参数：

- **aTo**：电话号码
- **aTemplate**：模板标识符。
- **aLanguageCode**：模板语言。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendMessageTemplate("34605889421", "hello_world", "en_US");
```

模板消息参数

模板可以包含参数，以下是带参数的默认模板示例

```
void SendSamplePurchaseFeedbackTemplate(string aName)
{
    TsgcWhatsApp_Send_Message_Template *oTemplate = new TsgcWhatsApp_Send_Message_Template();
    Try
    {
        oTemplate->Language->Code = "en_US";
        oTemplate->TemplateName = "sample_purchase_feedback";
        // ... header
        TsgcWhatsApp_Send_Message_Template_Component *oComponent = new TsgcWhatsApp_Send_Message_Template_Component();
        oComponent->_Type = wapctHeader;
        oTemplate->Components->Add(oComponent);
        TsgcWhatsApp_Send_Message_Template_Parameter *oParameter = new TsgcWhatsApp_Send_Message_Template_Parameter();
        oParameter->Image->Link = "https://www.esegece.com/images/esegece.png";
        oParameter->_Type = wapptImage;
        oComponent->Parameters->Add(oParameter);
        // ... body
        TsgcWhatsApp_Send_Message_Template_Component *oComponent = new TsgcWhatsApp_Send_Message_Template_Component();
        oComponent->_Type = wapctBody;
        oTemplate->Components->Add(oComponent);
        TsgcWhatsApp_Send_Message_Template_Parameter *oParameter = new TsgcWhatsApp_Send_Message_Template_Parameter();
        oParameter->Text = aName;
        oParameter->_Type = wapptText;
        oComponent->Parameters->Add(oParameter);
        whatsapp->SendMessageTemplate("107809351952205", oTemplate);
    }
    Finally
    {
        oTemplate->Free();
    }
    End;
}
```

模板消息上传图片

以下是一个模板示例，其中不使用图像链接，而是先将图像上传到服务器，然后设置文档的 ID。

```
oClient->OnMessage(TsgcWSConnection *Connection, const string Text)
void SendSamplePurchaseFeedbackTemplate(string aName)
{
    TsgcWhatsApp_Send_Message_Template *oTemplate = new TsgcWhatsApp_Send_Message_Template();
    Try
    {
        // ... first upload the file
        string vId = whatsapp->UploadMedia("c:\\images\\file.png", "image/png");

        // ... send message
        oTemplate->Language->Code = "en_US";
        oTemplate->TemplateName = "sample_purchase_feedback";
        // ... header
        TsgcWhatsApp_Send_Message_Template_Component *oComponent = new TsgcWhatsApp_Send_Message_Template_Component();
        oComponent->_Type = wapctHeader;
```

```

oTemplate->Components->Add(oComponent);
TsgcWhatsApp_Send_Message_Template_Parameter *oParameter = new TsgcWhatsApp_Send_Message_Template_Parameter()
oParameter->Image->id = vId;
oParameter->_Type = wapptImage;
oComponent->Parameters->Add(oParameter);
// ... body
TsgcWhatsApp_Send_Message_Template_Component *oComponent = new TsgcWhatsApp_Send_Message_Template_Component()
oComponent->_Type = wapctBody;
oTemplate->Components->Add(oComponent);
TsgcWhatsApp_Send_Message_Template_Parameter *oParameter = new TsgcWhatsApp_Send_Message_Template_Parameter()
oParameter->Text = aName;
oParameter->_Type = wapptText;
oComponent->Parameters->Add(oParameter);
whatsapp->SendMessageTemplate("107809351952205", oTemplate);
__Finally
oTemplate->Free();
End;
}

```

WhatsApp 接收消息和状态通知

订阅 [Webhook](#)，以接收您企业收到的消息及客户资料更新的通知。

每当触发事件发生时，WhatsApp Business Platform 会检测到该事件，并向您事先指定的 Webhook URL 发送通知。您可以收到两种类型的通知：

- **已接收消息**：此提醒让您知道您已收到消息。
- **消息状态和定价通知**：此提醒让您知道消息状态发生了变化，例如消息已被阅读或已送达。

已接收消息

每次收到新消息时，都会调用事件 **OnMessageReceived**，您可以在其中访问消息内容并将其标记为已读。

以下示例中，当收到新的文本消息时，将其回显给发送方。

```
void OnWhatsAppMessageReceived(TObject *Sender, const TsgcWhatsApp_Receive_Message *aMessage, ref bool aMarkAsRead)
{
    if (aMessage->Contacts->Count > 0)
    {
        string vTo = aMessage->Contacts->Contact[0]->WaID;
        if (aMessage->Messages->Count > 0)
        {
            if (aMessage->Messages->_Message[0]->_Type = wapmrtText)
            {
                vText = "ECHO ==> " + aMessage->Messages->_Message[0]->Text->Body;
                WhatsApp->SendMessageText(vTo, vText);
                aMarkAsRead = true;
            }
        }
    }
}
```

已发送的消息

WhatsApp 商业平台会发送通知，告知您与用户之间消息的状态。当消息发送成功后，您会在消息发送、投递和阅读时各收到一条通知。这些通知在您的应用程序中出现的顺序可能不反映消息状态的实际时间。如有必要，请查看时间戳以确定时间。

- **sent**：当企业在用户发起的对话中发送消息时（若该对话并非源自免费入口），将收到以下通知：
- **delivered**：当企业消息已送达，且该消息属于用户发起的对话（若该对话并非源自免费入口点），则收到以下通知：
- **read**：当用户阅读消息时，将收到以下通知。

每次收到新状态时，将调用 **OnMessageSent** 事件。

```
void OnWhatsAppMessageSent(TObject *Sender, const TsgcWhatsApp_Receive_Message *aMessage, TsgcWhatsAppSendMessage *aSendMessage)
{
    string vPhone := aMessage.MetaData.DisplayPhoneNumber
    if (aStatus = wapsmstSent) {
        DoLog("Message to " + vPhone + " sent.");
    } else if (aStatus = wapsmstDelivered) {
        DoLog("Message to " + vPhone + " delivered.");
    } else if (aStatus = wapsmstRead) {
        DoLog("Message to " + vPhone + " read.");
    } else {
        DoLog("Message to " + vPhone + " unknown status.");
    }
}
```

```
} }
```

WhatsApp 发送文件

所有 API 调用必须使用访问令牌进行认证。开发者可以使用在[应用仪表盘 > WhatsApp > 入门指南](#)中生成的访问令牌认证 API 调用

API 调用以字符串形式返回消息 ID。

当您使用 WhatsApp API 发送文件时，消息首先被上传到 WhatsApp 服务器，然后使用上传文件后返回的对象 ID 发送一条新消息。

图像消息

调用方法 **SendMessageImage** 并传入以下参数：

- **aTo**：电话号码
- **aFileName**：要发送的图像文件的完整文件名（含路径）。
- **aFileType**：
 - image/jpeg
 - image/png
- **aCaption**：图片标题（可选）。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendFileImage("34605889421", "c:\\images\\image.png", "image/png");
```

文档消息

调用 **SendMessageDocument** 方法并传入以下参数：

- **aTo**：电话号码
- **aFileName**：要发送的文档文件的完整文件名（含路径）。
- **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**：文档标题（可选）。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendFileDocument("34605889421", "c:\\MyDocuments\\invoice.pdf", "application/pdf");
```

音频消息

调用方法 **SendMessageAudio** 并传入以下参数：

- **aTo**：电话号码

- **aFileName**：要发送的音频文件的完整文件名（含路径）。
- **aFileType**：
 - audio/aac
 - audio/mp4
 - audio/mpeg
 - audio/amr
 - audio/ogg

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendFileAudio("34605889421", "c:\\Music\\audio.mp3", "audio/mp4");
```

视频消息

调用方法 **SendMessageVideo** 并传递以下参数：

- **aTo**：电话号码
- **aFileName**：要发送的视频文件的完整文件名（含路径）。
- **aFileType**：
 - video/mp4
 - video/3gp

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendFileVideo("34605889421", "c:\\Videos\\video.mp4", "video/mp4");
```

贴纸消息

调用方法 **SendMessageSticker**，并传入以下参数：

- **aTo**：电话号码
- **aFileName**：要发送的贴纸文件的完整文件名（含路径）。
- **aFileType**：
 - image/webp

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->SendFileSticker("34605889421", "c:\\Stickers\\MySicker.webp", "image/webp");
```

WhatsApp 下载媒体

如果您收到包含媒体文件链接的消息，可以使用 **DownloadMedia** 方法下载该媒体文件。

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient->WhatsappOptions->PhoneNumberId = "107809351952205";
oClient->WhatsappOptions->Token = "EAA040pgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient->DownloadMedia("38923878928822", "c:\\whatsapp\\media\\image.png");
```

要删除之前上传的媒体文件，只需调用 **DeleteMedia** 并将对象 ID 作为参数传入即可。

Telegram API

Telegram

Telegram 提供两种 API，一种是 **Bot API**，允许您使用机器人和 HTTPS 作为协议创建程序；另一种是 **Telegram API 和 TDLib**，允许您构建自定义 Telegram 客户端，功能远比 Bot API 强大。

sgcWebSockets 通过 **tdjson 库支持 TDLib**，这意味着您可以构建自己的 Telegram 客户端。TDLib 处理所有网络实现细节、加密和本地数据存储。TDLib 支持所有 Telegram 功能。

TDLib (Telegram 数据库库) 优势

- **跨平台**：可在 Windows、Android、iOS、MacOS、Linux 等平台上使用。
- **易于使用**：使用 JSON 消息在应用程序与 Telegram 之间进行通信。
- **高性能**：在 Telegram Bot API 中，每个 TDLib 实例可处理超过 24000 个机器人。
- **一致性**：TDLib 保证所有更新将按正确顺序传递。
- **可靠性**：TDLib 在慢速和不稳定的网络连接下保持稳定。
- **Secure**：所有本地数据都使用用户提供的加密密钥进行加密。
- **完全异步**：对 TDLib 的请求互不阻塞。响应将在可用时发送。

配置

Windows

TDLib 需要其他第三方库：OpenSSL 和 ZLib。这些库必须与 tdjson 库一同部署。

* Windows 版本需要 VCRuntime，可从 Microsoft 下载：<https://www.microsoft.com/en-us/download/details.aspx?id=52685>。如果安装后问题仍然存在，请尝试将以下 dll 复制到您的应用程序所在的同一文件夹中：VCRUNTIME140.dll 和 VCRUNTIME140_1.dll。

将以下库复制到您的应用程序所在的目录：

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

将 libtdjson.dylib 库部署到您的设备，并可使用 SetTDJsonPath 设置库的位置，示例：

如果部署到 "Contents\MacOS\"，必须在 TPath.GetDirectoryName(ParamStr(0)) 文件夹中设置路径。

OSXARM64

将 libtdjson.dylib 库部署到您的设备，并可使用 SetTDJsonPath 设置库的位置，示例：

如果部署到 "Contents\MacOS\"，必须在 TPath.GetDirectoryName(ParamStr(0)) 文件夹中设置路径。

Linux64

将库文件 libtdjson.so 部署到您的设备，并通过调用方法 SetTDJsonPath 设置库路径。

Android

将库 libtdjsonandroid.so 部署到您的设备。示例：如果部署 Android64 库，请在项目/部署中将 RemotePath 设置为 "library\lib\arm64-v8a"；如果是 Android32，则将 RemotePath 设置为 "library\lib\armeabi-v7a"。

iOS64

将库 libtdjson.a 复制到以下目录：

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

创建您的 Telegram 应用程序

若要获取 API id 并使用 Telegram API 开发自己的应用程序，您需要执行以下操作：

- 使用任意应用程序注册 Telegram。
- 登录您的 Telegram 核心：<https://my.telegram.org>。
- 前往 **API 开发工具** 并填写表单。
- 您将获得基本地址以及用户授权所需的 **api_id** 和 **api_hash** 参数。
- 目前每个号码只能连接一个 **api_id**。

这些值必须设置在 Telegram 组件的 **Telegram.API** 属性中。为了进行身份验证，您可以作为用户或机器人进行认证，有 2 个属性可用于登录 Telegram：

- **PhoneNumber**：如果您以用户身份登录，必须设置您的**手机号码**（含国际区号），例如：+34699123456。
- **BotToken**：如果以机器人身份登录，请在此属性中设置您的令牌（由 Telegram 提供）。
- **DatabaseDirectory**：允许您指定 tdlib 数据库的位置。留空则使用默认配置。

可以配置以下参数：

- **ApplicationVersion**：应用程序版本，示例：1.0
- **DeviceModel**：设备型号，例如：desktop
- **LanguageCode**：用户语言代码，例如：en。
- **SystemVersion**：操作系统版本，示例：windows。

您也可以使用 **SetTDJsonPath** 方法配置 tdjson 库的路径。只需在启动新的 Telegram 会话之前传入路径即可。

配置好 Telegram 组件后，可以将 Active 属性设为 true，程序将尝试连接到 Telegram。

示例代码

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "your api hash";
oTelegram->Telegram->API->ApiId = "your api id";
oTelegram->PhoneNumber = "your phone number";
oTelegram->ApplicationVersion = "1.0";
oTelegram->DeviceModel = "Desktop";
oTelegram->LanguageCode = "en";
oTelegram->SystemVersion = "Windows";
oTelegram->Active = true;
```

授权

库可以调用两个事件，分别用于获取认证码（通过 Telegram 应用程序而非短信发送）或提供密码。

OnAuthenticationCode

当 Telegram 向 Telegram 应用程序发送授权码时，将调用此事件，用户必须复制该代码并将其设置在此事件的 Code 参数中。

```
void OnAuthenticationCode(TObject *Sender, ref string Code)
{
    Code = InputBox("Telegram Code", "Introduce code", "");
}
```

OnAuthenticationPassword

当 Telegram 要求用户设置密码时调用此事件。

授权状态

授权开始后，您可以通过 **OnAuthorizationStatus** 事件检查授权状态，每次授权状态发生变化时都会调用此事件。Status 的部分值包括：

- authorizationStateWaitTdlibParameters
- authorizationStateWaitEncryptionKey
- authorizationStateWaitPhoneNumber
- authorizationStateWaitCode
- authorizationStateLoggingOut
- authorizationStateClosed
- authorizationStateReady

连接状态

连接启动后，您可以通过 **OnConnectionStatus** 事件检查连接状态，该事件在每次连接状态发生变化时调用。Status 的部分值如下：

- connectionStateConnecting
- connectionStateUpdating
- connectionStateReady

主要方法

TsgcTDLib_Telegram API 组件支持多种 Telegram 方法，以下列出最常用的方法。

方法	参数	描述
SendTextMessage	aChatId ：将发送消息的聊天 Id aText ：消息文本。 InlineKeyboard ：可选按钮（仅限机器人）。	向聊天发送文本消息
SendRichTextMessage	aChatId ：将发送消息的聊天 ID。 aText ：消息文本。 InlineKeyboard ：可选按钮（仅限机器人）。	向聊天发送富文本消息。Markdown 语法： • 粗体：**bold** • 斜体：<u><i>italic</i></u> • 删除线：~~strike~~ • 下划线：<u>underline</u> • 代码：##code##
SendDocumentMessage	aChatId ：要发送消息的聊天 ID aFilePath ：文档的完整文件路径 aInlineKeyboard ：可选按钮（仅限机器人）。	向聊天发送文档。
SendPhotoMessage	aChatId ：将发送消息的聊天 ID aFilePath ：照片的完整文件路径	向聊天发送照片。

	Width ：照片宽度。 Height ：照片的高度。 InlineKeyboard ：可选按钮（仅限机器人）。	
SendVideoMessage	aChatId ：将发送消息的聊天 ID aFilePath ：视频的完整文件路径 aWidth ：视频宽度。 Height ：视频的高度。 aDuration ：视频时长（秒）。 alInlineKeyboard ：可选按钮（仅限机器人）。	向聊天发送视频。
SendInvoiceMessage	aChatId ：要发送消息的聊天 ID。 alInvoice ：消息文本。 alInlineKeyboard ：可选按钮（仅限机器人）。	发送发票（仅当为机器人且在私人频道时可用）。
EditTextMessage	aChatId ：要发送消息的聊天的 ID aMessageId ：要修改的消息 ID Text ：消息文本。 InlineKeyboard ：可选按钮（仅限机器人）。 ShowKeyboard ：可选按钮（仅限 Bot）。	编辑消息的文本（或游戏消息的文本）
AddChatMember	aChatId ：要发送消息的聊天 ID aUserId ：用户的标识符。 aForwardLimit ：要转发给新成员的聊天中较早消息的数量；最多 100 条。对于超级群组 and 频道忽略此项。	向聊天添加新成员。成员不能被添加到私密或秘密聊天中。成员在聊天状态与服务器同步之前不会被添加。
AddChatMembers	aChatId ：将发送消息的聊天 Id aUserIds ：要添加到聊天中的用户标识符。	向聊天添加多个新成员。目前此选项仅适用于超级群组和频道。此选项不能用于加入聊天。在聊天状态与服务器同步之前，成员不会被添加。
GetChatMember	aChatId ：聊天标识符。 aUserId ：用户标识符。	返回聊天中某个成员的信息。
GetBasicGroupFullInfo	aGroupId ：基本组标识符	通过标识符返回基本群组的完整信息。
GetSupergroupMembers	aSuperGroupId ：超级群组或频道的标识符。 aSupergroupMembersFilter ：要返回的用户类型。默认为 null。 aOffset ：要跳过的用户数量。 aLimit ：返回用户的最大数量；最多 200。	返回超级群组或频道中成员或被封禁用户的信息。
JoinChatByInviteLink	aLink ：要导入的邀请链接；	如果可能，使用邀请链接将当前用户添加到聊天。在聊天状态与服务器同步之前，新成员不会被添加。
CreateNewSecretChat	aUserId ：用户的标识符。	创建一个新的秘密聊天。
CreateNewBasicGroupChat	aUserIds ：要添加到聊天的用户标识符。 aTitle ：新基本群组的标题	创建新的基础群组。
CreateNewSupergroupChat	aTitle ：新超级群组的标题 alsChannel ：若为 True，则创建频道聊天。 aDescription ：聊天描述。	创建新的超级群组或频道。

CreatePrivateChat	aUserId ：用户标识符。 aForce ：如果为 true，将在不发起网络请求的情况下创建聊天。在此情况下，除聊天类型、标题和照片外，其他所有关于聊天的信息可能不正确。	返回与指定用户对应的现有聊天
GetChats	aOffsetOrder ：从该聊天顺序开始返回聊天记录 aOffsetChatId ：从该聊天 ID 开始返回聊天记录 aLimit ：返回聊天记录的最大数量。	返回有序的聊天列表。聊天按 (order, chat_id) 对降序排序（以 Bot 身份登录时不可使用）
GetChat	aChatId ：聊天标识符	根据标识符返回聊天的相关信息
GetChatHistory	aChatId ：聊天标识符 aFromMessageId ：从此消息 ID 开始获取历史记录；使用 0 从最新消息开始获取结果。 aOffset ：指定 0 以从 from_message_id 精确获取结果，或指定最多 -99 的负偏移量以额外获取一些较新的消息。 aLimit ：返回的最大消息数	返回聊天中的消息，消息按时间倒序排列
GetUser	aUserId ：用户标识符	通过用户标识符返回用户信息。
AddProxyHTTP	aServer ：代理的服务器名称。 aPort ：代理端口号。 aUserName ：用于登录的用户名；可以为空。 aPassword ：登录密码；可为空。 aHTTPOnly ：若代理仅支持 HTTP 请求而不支持通过 HTTP CONNECT 方法建立透明 TCP 连接，则传入 true。	为网络请求添加 HTTP 代理服务器。可以在授权之前调用。
AddProxyMTPProto	aServer ：代理的服务器名称。 aPort ：代理端口号。 aSecret ：以十六进制编码的代理密钥。	为网络请求添加 MTPProto 代理服务器。可以在授权之前调用。
AddProxySocks5	aServer ：代理的服务器名称。 aPort ：代理端口号。 aUserName ：用于登录的用户名；可以为空。 aPassword ：登录密码；可为空。	为网络请求添加 Socks5 代理服务器。可以在授权之前调用。
EnableProxy	aid ：代理的 ID	启用代理。一次只能启用一个代理。可以在授权之前调用。
DisableProxy		禁用当前启用的代理。可在授权之前调用。
RemoveProxy	aid ：代理的 ID	移除代理服务器。可在授权之前调用。
GetProxies		返回当前设置的代理列表。可以在授权之前调用。
getChatSponsoredMessage	aChatId ：聊天的 ID	返回在聊天中显示的赞助消息；仅适用于频道聊天。如果聊天中没有赞助消息，则返回 404 错误。

ViewMessage	aSponsorChatId ：赞助商聊天的 ID aMessageId ：消息的 ID	通知 TDLib 用户正在查看消息。许多有用的活动取决于消息当前是否正在被查看。
注销		从 Telegram 退出登录。
TDLibSend	aRequest ：JSON 请求。	以 JSON 协议发送任意请求。

如何发送文本消息的示例

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "your api hash";
oTelegram->Telegram->API->ApiId = "your api id";
oTelegram->PhoneNumber = "your phone number";
oTelegram->Active = true;
...
oTelegram->SendTextMessage("1234", "My First Message from sgcWebSockets");
```

示例：如何发送"方法未实现"响应

您可以使用 TDLibSend 方法发送任意 JSON 消息，例如：加入 Telegram 聊天。

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "your api hash";
oTelegram->Telegram->API->ApiId = "your api id";
oTelegram->PhoneNumber = "your phone number";
oTelegram->Active = true;
...
oTelegram->TDLibSend("{\"@type\": \"joinChat\", \"chat_id\": \"1234\"}");
```

事件

OnBeforeReadEvent

当 Telegram API 组件收到 JSON 消息且尚未处理时调用此事件。如果您手动处理此事件或不希望组件处理该事件，请将 Handled 属性设置为 True。您也可以使用此事件记录所有消息。

OnMessageText

当收到新的文本消息时调用此事件，读取 MessageText 参数以访问消息文本属性。

- **ChatId**：聊天标识符。
- **MessageId**：消息标识符。
- **SenderId**：发送方标识符。
- **Text**：消息文本。

OnMessageDocument

当收到新的文档消息时调用此事件。访问 MessageDocument 以获取文档属性。

- **ChatId**：聊天标识符。
- **MessageId**：消息标识符。
- **SenderId**：发送方标识符（从 tdlb 1.7.+ 读取 SenderChat 和 SenderUser）。
- **FileName**：文档名称。
- **DocumentId**：文档标识符。
- **LocalPath**：本地文件的完整路径（如果存在）。
- **MimeType**：文档的 MIME 类型。
- **Size**：文档大小。
- **RemoteDocumentId**：远程文档标识符。

OnMessagePhoto

当收到新的照片消息时调用此事件。访问 `MessagePhoto` 以获取照片属性。

- **ChatId**：聊天标识符。
- **MessageId**：消息标识符。
- **SenderId**：发送方标识符（从 `tdlib 1.7.+` 读取 `SenderChat` 和 `SenderUser`）。
- **PhotoId**：照片标识符。
- **LocalPath**：本地文件的完整路径（如果存在）。
- **Size**：照片大小。
- **RemotePhotoId**：远程照片标识符。

OnVideoPhoto

当收到新的视频消息时触发此事件。可通过 `MessageVideo` 访问视频属性。

- **ChatId**：聊天标识符。
- **MessageId**：消息标识符。
- **SenderId**：发送方标识符（从 `tdlib 1.7.+` 读取 `SenderChat` 和 `SenderUser`）。
- **VideoId**：照片标识符。
- **LocalPath**：本地文件的完整路径（如果存在）。
- **Width**：视频宽度。
- **Height**：视频高度。
- **Duration**：视频时长（秒）。
- **Size**：视频大小。
- **RemoteVideoId**：远程照片标识符。

OnMessageSponsored

此事件在收到新的赞助消息时调用（在调用方法 `getChatSponsoredMessage` 之后）

- **SponsorChatId**：赞助商聊天标识符。
- **MessageId**：消息标识符。
- **Text**：消息文本。

OnNewChat

当收到新的聊天时调用此事件。

- **ChatId**：聊天标识符。
- **ChatType**：聊天类型（`chatTypeSupergroup`、`chatTypePrivate`.....）
- **Title**：聊天名称。
- **SuperGroupId**：如果是超级群组，则为群组 `Id`。
- **IsChannel**：返回是否为频道。

OnNewCallbackQuery

当收到新的传入回调查询时调用此事件；仅适用于机器人。

- **Id**：唯一查询标识符。
- **SenderId**：发送查询的用户标识符。
- **ChatId**：发送查询的聊天标识符。
- **MessageId**：发起查询的消息标识符。
- **ChatInstance**：唯一对应消息发送目标聊天的标识符。
- **PayloadData**：来自通用回调按钮的负载。
 - **Data**：附加到回调按钮的数据。

OnEvent

当 API 组件收到新事件时调用此事件。可用于处理 API 组件未实现的某些事件。

- **Event**：事件名称（例如：`updateOption`、`updateUser`.....）
- **Text**：完整 JSON 消息

OnException

处理 Telegram API 数据时发生任何异常，均会调用此事件。

属性

MyId：返回当前用户的用户标识符。

完整代码示例

请查看以下代码示例，演示了如何连接到 Telegram API、要求用户输入验证码（如果 Telegram API 需要）、在连接就绪时发送消息以及记录收到的文本消息。

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "your api hash";
oTelegram->Telegram->API->ApiId = "your api id";
oTelegram->PhoneNumber = "your phone number";
oTelegram->ApplicationVersion = "1.0";
oTelegram->DeviceModel = "Desktop";
oTelegram->LanguageCode = "en";
oTelegram->SystemVersion = "Windows";
oTelegram->Active = true;

void OnAuthenticationCode(TObject *Sender, ref string Code)
{
    Code = InputBox("Telegram Code", "Introduce code", "");
}

void OnMessageText(TObject *Sender, TsgcTelegramMessageText *MessageText)
{
    Log("Message Received: " + MessageText->Text);
}

void OnConnectionStatus(TObject *Sender, const string Status)
{
    if (Status == "connectionStateReady")
    {
        oTelegram->SendTextMessage("1234", "Hello Telegram!");
    }
}
```

Telegram | 发送带内联按钮的 Telegram 消息

Telegram API 允许您向用户发送带内联按钮的消息以选择答案（此选项仅适用于机器人）。

发送消息之前，请创建 **TsgcTelegramReplyMarkupInlineKeyboard** 类的实例，并为每个要创建的按钮调用 **AddButtonTypeCallback** 或 **AddButtonTypeUrl** 方法。

示例

创建一条新消息，询问用户是否喜欢该消息，并附上回答调查的链接。使用 **OnNewCallbackQuery** 事件处理响应。

```
TsgcTelegramReplyMarkupInlineKeyboard *oReplyMarkup = new TsgcTelegramReplyMarkupInlineKeyboard();
try
{
    oReplyMarkup->AddButtonTypeCallback("Yes", "I like it");
    oReplyMarkup->AddButtonTypeCallback("No", "I hate it");
    oReplyMarkup->AddButtonTypeUrl("Poll", "https://www.yoursite.com/telegram/poll");
    sgcTelegram->SendTextMessage("123456", "Do you like the message?", oReplyMarkup);
}
__finally
{
    oReplyMarkup->Free();
}

void OnNewCallbackQuery(TObject *Sender, TsgcTelegramCallbackQuery *CallbackQuery)
{
    if (CallbackQuery->PayloadData->Data == "I like it") then
    {
        ShowMessage("yes")
    }
    else
    {
        ShowMessage("no");
    }
}
```

Telegram | 发送带有按钮的机器人消息

Telegram API 允许您发送带有按钮的消息，以便向用户请求数据（此选项仅适用于机器人）。

在发送消息之前，创建 **TsgcTelegramReplyMarkupShowKeyboard** 类的实例，并为要创建的每个按钮调用 **AddButtonTypeRequestLocation**、**AddButtonTypeRequestPhoneNumber** 或 **AddButtonTypeText** 方法。

示例

创建一条新消息，要求用户提供 PhoneNumber

```
oReplyMarkup = new TsgcTelegramReplyMarkupShowKeyboard();
oReplyMarkup->AddButtonTypeRequestPhoneNumber("Give me your phone");
sgcTelegram->SendTextMessage("123456", "Please provide the information below", null, oReplyMarkup);
oReplyMarkup->Free();
```

Telegram | 发送加粗 Telegram 消息

您可以使用粗体、斜体等样式突出显示文本消息。使用 **SendRichTextMessage** 方法发送具有样式功能的文本消息，该方法会解析文本消息，并自动向 API Telegram 添加所需的实体。

Markdown 语法

- 粗体 [*****]

```
**This is Bold**
```

- 斜体 [*_*]

```
__This is Italic__
```

- Strike [~~-~~]

```
--This is Strike--
```

- 下划线 [~]

```
~~This is Underline~~
```

- 代码 [`#`]

```
##This is Monospace##
```

Telegram | 未找到聊天机器人（Bot）对应的聊天

当您以机器人身份登录时，无法使用 GetChats 方法，因此无法获取所有可用聊天。若这是您首次以机器人身份登录并尝试向已知聊天发送消息，您将收到以下错误：

```
{"@type":"error","code":5,"message":"Chat not found"}
```

解决方案是在发送 Telegram 消息之前调用 **GetChat** 方法，并将 **ChatId** 作为参数传入。获取聊天数据后，即可正常发送 Telegram 消息。

请注意，如果您之前从未收到过此聊天的机器人消息，则在发送消息之前只需在**第一次调用 GetChat**。如果您关闭应用程序后重新启动，则无需先调用 GetChat，因为聊天已保存在 Telegram 数据库中。

Telegram | 赞助消息

每次用户打开频道时，必须调用 `channels.getSponsoredMessages` 以接收该频道的可用赞助消息。结果必须缓存 5 分钟。

显示赞助消息

赞助消息必须显示在频道中所有其他帖子的下方，用户继续向下滚动，超过最后一条消息后才能看到。from_id 字段中指定的推广频道或机器人必须显示为消息的作者。消息底部还应包含以下按钮之一：

- **查看机器人**：如果正在推广一个机器人，点击按钮必须打开与该机器人的聊天。如果指定了 start_param，应用程序必须使用深度链接机制打开机器人。
- **查看频道**：如果某个频道正在被推广，点击按钮必须打开该频道。
- **查看帖子**：如果某个频道正在推广且指定了 channel_post，则点击按钮必须打开该特定频道帖子。

一旦整个文本（不包括按钮）显示在屏幕上，必须使用该赞助消息的 random_id 调用 **ViewMessage** 方法。

获取赞助消息

向频道发送请求以询问是否有可用的赞助消息，只需调用方法 **GetChatSponsoredMessage**。

```
TsgcTDLlib_Telegram *oTelegram = new TsgcTDLlib_Telegram();
oTelegram->Telegram->API->ApiHash = "ABCDEFGHijklmn";
oTelegram->Telegram->API->ApiId = "1234";
oTelegram->PhoneNumber = "008745744155";
oTelegram->Active = true;
oTelegram->getChatSponsoredMessage("100");
```

若聊天中有赞助消息，将调用 **OnMessageSponsored** 事件，并携带赞助消息的内容。若没有消息，则返回 404 错误。

```
private void(TObject *Sender, TsgcTelegramMessageSponsored *MessageSponsored)
{
    DoLog(MessageSponsored->Text);
}
```

在赞助消息展示给用户后，调用 **ViewMethod** 方法。

```
oTelegram->ViewMessage("100", "54653256245");
```

Telegram | 发送 Telegram 发票消息

如果您的机器人支持内联模式，用户还可以通过机器人将发票发送到其他聊天，包括与其他用户的一对一聊天。

发票消息具有产品的照片和描述以及醒目的"支付"按钮。点击此按钮将在 Telegram 应用中打开一个特殊的支付界面

机器人可以使用方法 **SendInvoiceMessage** 将发票作为消息发送。

```
private void SendInvoice()
{
    TsgcTelegramSendInvoice *oInvoice = new TsgcTelegramSendInvoice();
    Try
    {
        oInvoice->Title = "Invoice Title Test";
        oInvoice->Description = "Description Invoice Test";
        oInvoice->Invoice->Currency = 'EUR';
        oInvoice->Invoice->Total = 800;
        oInvoice->Invoice->IsTest = True;
        oInvoice->Invoice->Payload := "payload";
        oInvoice->Invoice->ProviderToken := "provider_token";
        oInvoice->Invoice->ProviderData := "provider_data";

        sgcTelegram->SendInvoiceMessage("3284239872", oInvoice);
    }
    finally
    {
        oInvoice->Free();
    }
}
```


Telegram | 获取超级群组成员

Telegram API 允许您获取超级群组成员的信息。使用方法 **GetSuperGroupMembers** 获取超级群组或频道的成员或封禁用户信息。仅当 `SupergroupFullInfo.can_get_members` 为 `true` 时方可使用；此外，某些过滤条件可能需要管理员权限。

默认情况下，该方法返回群组的所有成员，但您可以使用 `Filter` 参数过滤返回的成员。可用的过滤参数如下：

tsgmFilterNone

默认值，表示成员不被过滤。

tsgmFilterAdministrators

返回创建者 和管理员。

tsgmFilterBanned

返回超级群组或频道中的封禁用户列表；只有管理员可以使用。
您可以使用参数 `aSuperGroupMembersQuery` 进行查询搜索。

tsgmFilterBots

返回超级群组或频道的机器人成员。

tsgmFilterContacts

返回超级群组或频道的成员中属于该用户联系人的列表。
您可以使用参数 `aSuperGroupMembersQuery` 进行查询搜索。

tsgmFilterMention

返回可在超级群组中提及的用户。

tsgmFilterRecent

按逆时间顺序返回最近活跃的用户。

tsgmFilterRestricted

返回受限超级群组成员；只能由管理员使用。
您可以使用参数 `aSuperGroupMembersQuery` 进行查询搜索。

tsgmFilterSearch

用于通过（字符串）查询搜索超级群组或频道成员。
您可以使用参数 `aSuperGroupMembersQuery` 进行查询搜索。

您可以使用 `OnEvent` 回调并通过 `event = "chatMembers"` 过滤来读取结果。

```
Telegram->GetSupergroupMembers(1452979380);
```

```
private void OnTelegramEvent(TObject *Sender, const string Event, const string Text)
{
    if (Event == "chatMembers")
    {
        ReadJSON(Text);
    }
}
```

Telegram | 添加 Telegram 代理

Telegram 客户端可配置为使用代理。目前，Telegram 支持 3 种类型的代理：

1. HTTP
2. MTProto
3. Socks5

添加代理

要配置 HTTP 代理，首先需要将代理添加到 Telegram 配置中，只需调用 **AddProxyHTTP**，如果成功，将返回一条包含新添加代理的消息。将代理添加到列表后，调用 **EnableProxy** 并传入确认消息中收到的代理 ID。

```
Telegram->AddProxyHTTP("8.8.8.8", 8080, "", "", true);  
// ... read the confirmation message and save the ID of the proxy.  
Telegram->EnableProxy(2);
```

删除代理

调用 **RemoveProxy** 方法，并传入要删除的代理 ID。

Telegram | 注册 Telegram 用户

在 Telegram 注册新用户的过程非常简单，您需要 API Id、API Hash 以及新账户的手机号码。

配置 Telegram 客户端：

- API Id
- API Hash
- 新 Telegram 账号的电话号码。

启动客户端后，将向手机发送新验证码，客户端会请求输入 Telegram 验证码，若验证码正确，将调用 OnRegisterUser 事件。在此事件中设置用户的名字和姓氏，注册即完成。

```
TsgcTDLib_Telegram *oTelegram = new TsgcTDLib_Telegram();
oTelegram->Telegram->API->ApiHash = "ABCDEFGHJKLMN";
oTelegram->Telegram->API->ApiId = "1234";
oTelegram->PhoneNumber = "008745744155";
oTelegram->Active = true;

void OnTelegramAuthenticationCode(TObject *Sender, ref string Code)
{
    Code = "code sent to phone";
}

void OnTelegramRegisterUser(TObject *Sender, ref string FirstName, ref string LastName)
{
    FirstName = "first name";
    LastName = "last name";
}
```

RCON

RCON

Source RCON 协议是一种基于 TCP/IP 的通信协议，由 Source Dedicated Server 使用，允许通过"远程控制台"（即 RCON）向服务器发出控制台命令。RCON 最常见的用途是允许服务器所有者在没有直接访问运行服务器的机器的情况下控制其游戏服务器。

配置

RCON_Options 允许您配置以下属性：

- **Host**：服务器远程地址。
- **Port**：服务器监听端口。
- **Password**：用于对服务器进行身份验证的密钥字符串。

连接

使用 **Active** 属性连接/断开服务器。

当 Active 设置为 True 时，客户端尝试连接到服务器，如果可以连接，将尝试使用提供的密码进行身份验证。服务器将向认证请求发送响应。**OnAuthenticate** 事件将被调用，您可以通过 Authenticate 参数读取认证是否成功。

发送命令

使用 **ExecCommand** 方法向服务器发送命令。响应将在 **OnResponse** 事件中获取。

```
TsgcLib_RCON oRCON = new TsgcLib_RCON();
oRCON->RCON_Options->Host = "127.0.0.1";
oRCON->RCON_Options->Port = 25575;
oRCON->RCON_Options->Password = "test";
oRCON->Active = true;

void OnAuthenticate(TObject *Sender, bool Authenticated, const TsgcRCON_Packet *aPacket)
{
    if (Authenticated == true)
    {
        DoLog("#authenticated");
    }
    else
    {
        DoLog("#not authenticated");
    }
}

void OnResponse(Object *Sender, const string aResponse, const TsgcRCON_Packet *aPacket)
{
    DoLog(aResponse);
}
```

CryptoHopper

CryptoHopper

CryptoHopper 是一款自动化加密货币交易机器人，可自动化交易和管理比特币、以太坊、莱特币等数字资产的投资组合。

配置

需要开发者账号，获批后即可开始创建新应用。该 API 使用 OAuth2 进行身份验证，您可以从应用中获取 **client_id** 和 **client_secret**。

```
TsgcHTTP_Cryptohopper oCryptoHopper = new TsgcHTTP_Cryptohopper();
oCryptoHopper->CryptoHopperOptions->OAuth2->ClientId = "client_id";
oCryptoHopper->CryptoHopperOptions->OAuth2->ClientSecret = "client_secret";
oCryptoHopper->CryptoHopperOptions->OAuth2->LocalIP = "127.0.0.1";
oCryptoHopper->CryptoHopperOptions->OAuth2->LocalPort = 8080;
oCryptoHopper->CryptoHopperOptions->OAuth2->Scope->Text = "read,notifications,manage,trade";
```

方法

CryptoHopper 使用 HTTPS 作为协议向 API 发送请求。某些方法需要身份验证（下订单、获取用户数据等），而其他方法则是公开的（例如获取交易所数据）。

函数返回 CryptoHopper 的响应，如果发生任何错误，将引发异常。

Hoppers

管理基本 Hopper 操作。

方法	参数	描述
GetHoppers		获取用户的 Hopper 列表。
CreateHopper	aBody ：配置 JSON 文本。	创建新的 Hopper。
GetHopper	aid ：hopper id	检索料斗
DeleteHopper	aid ：hopper id	删除 Hopper
UpdateHopper	aid ：hopper ID aBody ：配置 JSON 文本。	更新 Hopper

订单

管理您 Hopper 的订单。

方法	参数	描述
GetOpenOrders	ald : hopper id	检索 hopper 的所有未结订单。
CreateNewOrder	ald : hopper id aOrder : TsgcHTTPCTHOrder 的实例	创建新的买入或卖出订单。对于卖出, 建议使用 sell 端点。
PlaceMarketOrder	ald : hopper id aOrderSide : cthosBuy 或 cthosSell。 aCoin : 币种名称, 例如: EOS aAmount : 订单大小。	下达市价订单。
PlaceLimitOrder	ald : hopper id aOrderSide : cthosBuy 或 cthosSell。 aCoin : 币种名称, 例如: EOS aAmount : 订单大小。 aPrice : 限价价格。	下限价单
DeleteOrder	ald : hopper id aOrderId : 订单 ID	删除所选料斗的订单。
DeleteAllOrders	ald : hopper id	删除所选 hopper 的所有未完成订单。
GetOpenOrder	ald : hopper id aOrderId : 订单 ID	按 ID 获取 hopper 中的未结订单。
CancelOrder	ald : hopper id aOrderId : 订单 ID	取消一个未成交的订单。

仓位

管理您的 Hopper 持仓。

方法	参数	描述
GetPosition	ald : hopper id	获取 hopper 的持仓。

交易

来自您的 Hopper 的交易历史记录。

方法	参数	描述
GetTradeHistory		获取 Hopper 的交易历史。

GetTradeHistoryById	aId : hopper ID aTradeId : 交易 ID	根据 hopper ID 获取一笔交易。

交换机

从 CryptoHopper 获取可用交易所信息

方法	参数	描述
GetExchange		获取 Cryptohopper 上所有可用的交易所。
GetAllTickers	aExchange : 交易所名称	获取所有交易对的行情
GetMarketTicker	aExchange : 交易所名称 aPair : 交易对名称	获取市场交易对的行情。
GetOrderBook	aExchange : 交易所名称 aPair : 交易对名称 aDepth : 订单簿深度	获取指定交易所、市场及订单簿深度的订单簿信息。

Webhooks

来自您的 Hopper 的交易历史记录。

方法	参数	描述
CreateWebhook	aURL : Webhook URL aMessageTypes : 以逗号分隔的消息类型。	更新或创建 Webhook
DeleteWebhook	aURL : webhook URL	删除现有的 Webhook。

信号

向 CryptoHopper API 发送信号。

方法	参数	描述
SendSignal	aSignal : 包含发送信号所需所有字段的类。	发送信号

SendTestSignal	aSignal : 包含发送信号所需所有字段的类。	发送测试信号
GetSignalStats	aSignalId : 信号 ID。 aExchange : 可选, 交易所名称。	获取部分信号统计数据。

如何更新 Cryptohopper 配置

使用 UpdateHopper 方法更新 Hopper 配置。该方法已重载, 您可以传入 JSON 字符串, 也可以使用 TsgcHTTPCTHopper 对象并通过属性启用或禁用 Hopper 属性。

```
public string EnableHopper()
{
    TsgcHTTPCTHopper *oHopper = new TsgcHTTPCTHopper();
    try
    {
        oHopper->Enabled = 1;
        result = Cryptohopper->UpdateHopper("1234", oHopper);
    }
    __finally
    {
        oHopper->Free();
    }
}
```

如何配置 Webhook

Webhook 允许您在 hopper 中发生某些事情时接收通知。Webhook 需要一个公共 HTTPs 服务器, 用于监听 cryptohopper 发送的所有消息的 URL 地址。公共服务器需要受 SSL 证书保护 (不允许使用自签名证书)。

首先, 您必须创建一个 webhook, 因此请配置 Cryptohopper 客户端的 Webhook 属性, 设置服务器侦听的 Host 和 Port, 然后在 SSLOptions 属性中配置证书。

示例: 公共 IP 地址为 1.1.1.1, 监听端口为 443。证书以 PEM 文件存储, 文件名为 sgc.pem, 无密码。

```
/* 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 是一个 WebRTC JavaScript 库，适用于点对点应用程序（屏幕共享、音视频会议、文件共享、媒体流等）。

配置

RTCMultiConnection 需要一个 WebSocket 服务器用于信令，因此将 RTCMultiConnection 的 server 属性链接到一个 WebSocket 服务器（例如 [TsgcWebSocketHTTPServer](#)）。以下是您必须配置的属性。

服务器

Host：WebSocket 服务器的公网 IP 地址或 DNS 名称。
Port：WebSocket 服务器的监听端口。

IceServers

这是 ICE 服务器（STUN/TURN）的配置，用于实现节点间的通信。示例：

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

VideoResolution

在此处您可以配置视频会议的视频分辨率，分辨率越高，连接所需的带宽越大。

HTMLDocuments

为每个应用程序配置提供此内容的 HTML 页面名称。

示例：如果服务器运行在网站 [www.webrtc.com](#) 的 8443 端口，且 `HTMLDocuments.VideoConferencing = /RTCMultiConnection-VideoConferencing.html`，则访问视频会议的 URL 为

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

WebRTC 需要安全连接（HTTPS），因此需要使用证书，详情请阅读 [服务器 SSL](#)。

应用程序

名称	描述
视频会议	使用网状网络模型的多用户（多对多）视频聊天。
屏幕共享	使用星形拓扑的多用户（一对多）屏幕共享。
VideoBroadcasting	使用星形拓扑的多用户（一对多）视频广播。

WebPush

[RFC 8030](#)

[RFC 8291](#)

WebPush 协议由 **RFC 8030**（使用 HTTP Push 投递）和 **RFC 8291**（消息加密）定义。

Web Push 是一种用于向网页浏览器**投递推送通知**的**标准化协议**。它使用 Push API，这是一种标准的 Web API，使网站能够注册并接收推送消息。Push API 允许网站向用户的浏览器发送推送消息，即使用户当前未在浏览该网站。

要使用 Web Push，网站首先需要**从用户的浏览器获取推送订阅**。订阅由唯一的端点 URL 和加密密钥组成。端点 URL 是网站可以用来向用户浏览器发送推送消息的 URL，加密密钥用于加密和解密推送消息。

一旦网站**获得推送订阅**，就可以通过向端点 URL 发送 HTTP 请求向用户浏览器**发送推送消息**。推送消息以称为 Web Push Protocol Message 的特殊格式发送，由一组头部和有效载荷组成。头部包含加密密钥和消息的 TTL（生存时间）等信息，有效载荷包含消息的实际内容。

当**用户的浏览器收到推送消息**时，它**首先使用加密密钥解密消息**。然后它**向用户显示通知**，以及用户可以采取的任何其他操作，例如关闭通知或打开网站。

为确保推送消息的安全性和隐私性，Web Push 使用端到端加密，并要求推送订阅通过安全连接（如 HTTPS）获取。此外，该协议提供了验证推送消息发送方身份的机制，并防止滥用（如限制网站可向用户发送的推送消息数量）。

组件

有 2 个组件支持 WebPush：

- **[TsgcWSAPIServer_WebPush](#)**：在服务器端实现 WebPush 协议，允许您请求用户权限、注册订阅、发送通知等。此组件已封装了发送通知的 WebPush 客户端。
- **[TsgcWebPush_Client](#)**：在客户端实现 WebPush 协议，允许您通过桌面和移动 Web 向用户发送通知。如果您已拥有密钥和端点，只需向已订阅的客户端发布 WebPush 消息，此组件非常实用。

TsgcWSAPIServer_WebPush

TsgcWSAPIServer_WebPush 组件。

简介

TsgcWSServer_API_WebPush 是一个提供 WebPush 订阅处理功能的组件。WebPush 是一种向在浏览器中运行的 Web 应用程序实时推送通知的协议。此组件可用于管理订阅和向已订阅的客户端发送通知。以下是 TsgcWSServer_API_WebPush 类提供的属性、事件和方法，以及演示其使用方法的代码示例。

配置

1. 使用 **Server** 属性将 **TsgcWSServer_API_WebPush** 附加到 WebSocket 服务器。
2. 在 **WebPush.VAPID** 属性中配置**公钥**和**私钥**。（注册用户可下载用于在 Windows 上生成 VAPID 密钥的可执行文件。）
3. 需要部署 **OpenSSL 3.0.0 版本**
4. 在 **WebPush.Endpoints** 属性中，您可以定义自己的端点来处理 WebPush 订阅；默认情况下，访问 `"/sgcWebPush.html"` 端点将显示一个简单的网页，允许您订阅 WebPush 通知。
5. 启动服务器并访问已配置的端点进行测试。

参考

- 属性 — 组件上提供 3 个属性。
- 方法 — 3 个公开方法。
- 事件 — 3 个可处理的事件。

TsgcWebPush_Client

TsgcWebPush_Client 是一个允许您在获取订阅详情后发送通知的类。

简介

TsgcWebPush_Client 是一个允许您在获取订阅详情后发送通知的类。

以下示例展示了如何使用 WebPush 客户端，根据 WebPush 订阅中的端点、公钥和身份验证密钥发送通知。

```
void SendWebPushNotification()
{
    TsgcHTTP_API_WebPush_PushSubscription* oSubscription = new TsgcHTTP_API_WebPush_PushSubscription();
    try
    {
        oSubscription->Endpoint = "endpoint";
        oSubscription->PublicKey = "public key";
        oSubscription->AuthSecret = "authentication secret";
        TsgcHTTP_API_WebPush_Client* oWebPush = new TsgcHTTP_API_WebPush_Client(NULL);
        try
        {
            oWebPush->VAPID->PEM->PrivateKey->Text = "private_key_pem";
            oWebPush->VAPID->DER->PrivateKey = "private_key";
            oWebPush->VAPID->DER->PublicKey = "public_key";
            oWebPush->SendNotification(oSubscription, "{\"title\": \"eSeGeCe Notification\", \"body\": \"Hell\"}");
        }
        __finally
        {
            oWebPush->Free();
        }
    }
    __finally
    {
        oSubscription->Free();
    }
}
```

参考

- 属性 — 组件上提供 3 个属性。
- 方法 — 1 个公共方法。
- 事件 — 0 个可处理的事件。

扩展

WebSocket 协议被设计为可扩展的。WebSocket 客户端可以请求扩展，WebSocket 服务器可以接受客户端请求的部分或全部扩展。

支持的扩展：

1. [Deflate-Frame](#)：压缩 WebSocket 帧。
2. [PerMessage-Deflate](#)：压缩 WebSocket 消息。

Extensions | PerMessage-Deflate

PerMessage 是 WebSocket 协议扩展，如果服务器和客户端都支持该扩展，则双方均可压缩传输的消息：

- 使用 Deflate 作为压缩方法。
- 压缩仅适用于应用程序数据（控制帧 和标头不受影响）。
- 服务器和客户端可以选择压缩哪些消息。

最大窗口位数

此扩展允许自定义 LZ77 算法使用的服务器端和客户端滑动窗口大小（范围 8 到 15）。该值越大，查找并消除重复内容的可能性越高，但会消耗更多内存和 CPU 周期。默认值为 15。

No Context Take Over

默认情况下，之前的消息会被用于压缩和解压缩。若消息相似，这有助于提高压缩率。若启用此选项，每条消息将仅使用其自身数据进行压缩。默认情况下，此选项处于禁用状态。

MemLevel

此值不在服务器和客户端之间协商。当设置为 1 时，使用内存最少，但会减慢压缩算法并降低压缩比；当设置为 9 时，使用最多内存并提供最佳性能。默认情况下，设置为 1。

** 与 Rad Studio XE2 一同提供的 Indy 版本因 zlib 版本与初始化函数不匹配而引发异常。要解决此问题，只需将您的 Indy 版本更新至最新版即可。*

扩展 | Deflate-Frame

这是一个 WebSocket 协议扩展，允许压缩使用 WebSocket 协议发送的帧，受 Chrome 或 Safari 等 WebKit 浏览器支持。此扩展在服务器和客户端组件上均受支持。

此扩展已弃用。

** 与 Rad Studio XE2 一同提供的 Indy 版本因 zlib 版本与初始化函数不匹配而引发异常。要解决此问题，只需将您的 Indy 版本更新至最新版即可。*

MCP

MCP（模型上下文协议）是一种标准化协议，旨在定义和管理语言模型与外部系统之间的**上下文信息交换**。

它允许结构化、安全和动态的通信，使 AI 模型能够**理解、扩展并与其本机环境之外的外部工具或数据源交互**。

MCP 为传递上下文元数据提供了一个**统一接口**，使模型能够以可预测且可互操作的方式解释指令、检索数据并执行任务，其精神类似于 *WebAuthn* 对身份验证流程的标准化。

用途

MCP 的开发旨在解决模型、客户端和工具之间**上下文碎片化和集成不一致**的挑战。

通过 MCP，任何合规的客户端都可以通过共享的、定义明确的结构与任何兼容 MCP 的模型或服务进行通信。

典型用例包括：

- 将**上下文元数据**（用户会话、应用状态、权限等）附加到模型查询。
- 动态交换**函数或工具定义**。
- 跨模型会话管理有**状态交互**。
- 在 LLM、SDK 与后端服务之间建立**安全互操作性**。

组件

- **TsgcWSAPIServer_MCP**：该组件提供了一种简单而强大的 MCP 服务器实现方案，具备工具、提示和资源请求、身份验证、通过事件进行流量控制等功能。
- **TsgcWSAPIClient_MCP**：该组件为实现 MCP 客户端提供了简单而强大的解决方案，具备工具、提示和资源请求等功能。

TsgcWSAPIServer_MCP

TsgcWSAPIServer_MCP 组件。

简介

TsgcWSAPIServer_MCP 通过 `sgcWebSockets` HTTP 服务器端点暴露模型上下文协议 (MCP)。该组件将传入的 HTTP 请求与 `TsgcAI_MCP_Server` 引擎桥接, 使 MCP 兼容的客户端可以协商会话、枚举提示词/资源/工具, 并通过 JSON-RPC 风格的调用调用工具。

配置

- **放置组件**：`TsgcWSAPIServer_MCP` 组件必须连接到 HTTP 服务器, [TsgcWebSocketHTTP_Server](#) 或 [TsgcWebSocketServer_HTTPAPI](#), 使用 `Server` 属性。
- **选择端点**：如果默认的 `/mcp` 路径需要更改, 请调整 `EndpointOptions.Endpoint`。文档等于该路径的请求将被视为 MCP 调用；其他请求回退到父 API 的资源处理。
- **配置 MCP 设置**：使用 `MCPOptions.ServerInfo` 为服务器命名/设置版本, 使用 `MCPOptions.SessionTimeout` 控制会话生命周期, 使用 `MCPOptions.AuthenticationOptions` 要求在处理任何 JSON-RPC 调用之前提供 API 密钥或自定义 HTTP 头。
- **注册提示词、工具、资源和资源模板**：填充组件公开的只读列表 `Tools`、`Prompts`、`Resources` 和 `ResourceTemplates`。
- **处理事件**：连接下方描述的已发布事件, 以便为会话生命周期、提示/资源解析或工具执行插入业务逻辑。可以拦截异常以自定义 HTTP 响应代码。

内置 HTTP 端点（无需配置）

当组件附加到 HTTP 服务器时, 以下端点和行为将与 `/mcp` 一起自动提供, 无需启用任何新属性。这使其能够与基于 Web 的 MCP 连接器 (如 [claude.ai](#)) 开箱即用兼容, 这些连接器要求通过 CORS 支持带动态客户端注册的 OAuth 2.1。

- **CORS 支持与来源反射**。对任何已处理路径的 `OPTIONS` 预检返回 `204 No Content`, 带有以下标头：
 - `Access-Control-Allow-Origin`：回显请求的 `Origin` 标头, 若缺失则为 `*`。
 - `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`所有非 `OPTIONS` 响应也包含 `Access-Control-Allow-Origin`、`Access-Control-Expose-Headers` 和 `Strict-Transport-Security`。
- **401 Unauthorized（附带 `www-Authenticate`）**。当 `MCPOptions.AuthenticationOptions.ApiKey`（或 `CustomHeader`）启用且请求没有或凭证错误时, 服务器返回 `401` 并附带以下形式的标头：
`Bearer realm="MCP", resource_metadata="https://<host>/.well-known/oauth-protected-resource"`
- **OAuth 2.1 发现和流程端点**。以下端点与 `/mcp` 在同一 HTTP 服务器上提供, 支持 PKCE (S256)、授权码和刷新令牌授权以及动态客户端注册：

路径	规范	行为
<code>GET /.well-known/oauth-</code>	RFC 8414	JSON 授权服务器元数据。公告 PKCE (S256)、 <code>authorization_code</code> + <code>refresh_token</code> 授权类型、公共客户

路径	规范	行为
<code>authorization-server</code>		端 + <code>client_secret_basic</code> 认证方式、在 <code>/oauth/register</code> 处的动态注册，以及 <code>mcp</code> 作用域。
<code>GET /.well-known/oauth-protected-resource</code>	RFC 9728	JSON，将 MCP 端点描述为受保护资源，并将本服务器指定为授权服务器。
<code>POST /oauth/register</code>	RFC 7591 (DCR)	接受 JSON 客户端注册体（ <code>client_name</code> 、 <code>redirect_uris</code> ）。颁发生成的 <code>client_id</code> （无 <code>client_secret</code> ，公共客户端）。返回 <code>201</code> 及已注册的客户端信息响应。
<code>GET /oauth/authorize</code>	RFC 6749 + RFC 7636	呈现 HTML 授权同意页面。必填查询参数： <code>client_id</code> 、 <code>redirect_uri</code> 、 <code>response_type=code</code> 、 <code>code_challenge</code> 、 <code>code_challenge_method=S256</code> 、 <code>state</code> 、 <code>scope</code> 。
<code>POST /oauth/authorize</code>	RFC 6749	处理授权同意表单。用户必须输入与 <code>MCPOptions.AuthenticationOptions.ApiKey.Value</code> 相同的值以批准。批准后重定向至携带 <code>?code=...&state=...</code> 的 <code>redirect_uri</code> 。拒绝或密钥错误时，重定向携带 <code>?error=access_denied&state=...</code> ，或重新呈现表单。
<code>POST /oauth/token</code>	RFC 6749 + RFC 7636	表单编码的请求体。支持 <code>grant_type=authorization_code</code> （验证 PKCE： <code>BASE64URL(SHA256(code_verifier)) == code_challenge</code> ）和 <code>grant_type=refresh_token</code> 。颁发有效期 1 小时的 Bearer 访问令牌及刷新令牌。授权码在 60 秒后过期且只能使用一次。

- **Bearer 令牌验证已扩展。**当 `MCPOptions.AuthenticationOptions.ApiKey.Enabled` 为 `True` 时，如果 `<x>` 匹配已配置的静态 `ApiKey.Value`，或者匹配当前有效的 OAuth 颁发的访问令牌（通过上述流程颁发），则 `/mcp` 请求上的 `Authorization: Bearer <x>` 标头将被接受；任一条件满足即可。
- **内存存储。**OAuth 客户端、授权码和令牌仅保存在内存中，服务器重启后将丢失。
- **通过 `OnMCPHttpRequest` 覆盖。**绑定到 `OnMCPHttpRequest` 的处理程序可以通过设置 `Handled := True` 来短路上述任何行为。

参考

- 属性 — 组件上有 5 个可用属性。
- 方法 — 9 个公开方法。
- 事件 — 共 12 个可处理的事件。

MCP 服务器 | 会话

服务器与客户端之间成功完成初始化后，将创建一个新会话。会话始终随客户端和服务器之间的请求和响应一同发送，以标识请求/响应流中使用的会话。使用 HTTP 传输时，会话的生命周期为单次 HTTP 请求和响应，即响应发送后，会话即关闭。

会话对象在每个事件上传递，要知道会话何时被创建或删除，请使用以下事件：

- **OnMCPSessionNew**：每次创建新会话 ID 时，将调用此事件。
- **OnMCPSessionEnd**：会话结束时调用此事件。

OnMCPSessionNew

当新会话创建时调用此事件。您可以访问 Session 参数以获取会话 ID、会话到期时间等信息。

OnMCPSessionEnd

当会话结束时调用此事件。

```
void __fastcall TForm1::OnMCPSessionEnd(TObject *Sender, const TsgcAI_MCP_Session *aSession)
{
    printf("#session_end: %s\n", aSession->Id.c_str());
}
void __fastcall TForm1::OnMCPSessionNew(TObject *Sender, const TsgcAI_MCP_Session *aSession)
{
    printf("#session_new: " + aSession->Id);
}
```

MCP 服务器 | 工具

TsgcWSAPIServer_MCP 是通过 **sgcWebSockets** 公开模型上下文协议（MCP）服务器功能的高级组件。当 **MCP 客户端** 根据 **MCP 服务器工具规范** 请求工具时，此组件编排 HTTP 网关、会话生命周期和 JSON-RPC 序列化，使您的 Delphi 代码能够专注于实现业务逻辑。

命中 **EndpointOptions.Endpoint** 中定义的 MCP 端点的传入 HTTP 请求会被 **TsgcWSServer_API_MCP** 拦截。该组件创建（或恢复）一个 MCP 会话，将请求转发给底层 **TsgcAI_MCP_Server** 引擎，最终将 JSON-RPC 响应写回 HTTP 客户端，同时保留规范要求的 **mcp-session-id** 标头。

工具列表

服务器在 **TsgcAI_MCP_ToolsList** 中维护一个内存中的工具目录。工具按名称保证唯一，公开描述并提供类似 JSON Schema 的输入描述，该描述在响应规范的 **tools.list** 方法时发出。当客户端调用 **tools.list** 时，**TsgcWSAPIServer_MCP** 加载请求、序列化当前目录并以 200 HTTP 状态代码发回。

发布简单计算器工具的示例代码：

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    TsgcAI_MCP_Tool* oTool;
    // Add a tool named "math.add"
    oTool = MCPServer->Tools->AddTool("math.add", "Adds two numbers");
    // Define input schema properties
    oTool->InputSchema->Properties->AddProperty("a", true, aimcpjtNumber, "Left operand");
    oTool->InputSchema->Properties->AddProperty("b", true, aimcpjtNumber, "Right operand");
}
```

发送给客户端的符合规范的 **tools.list** 响应将反映 MCP JSON 模式负载。例如：

```
{
  "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" }
          }
        }
      }
    ]
  }
}
```

工具请求

当客户端发出 **tools.call** JSON-RPC 请求时，**TsgcWSAPIServer_MCP** 会将强类型请求对象（包括工具名称和提供的参数）填充完整，然后触发 **OnMCPRequestTool** 事件。您的处理程序填充响应负载，该负载随后会与成功的 HTTP 状态码一起序列化并返回给客户端。

典型的处理程序如下所示：

```
void __fastcall TMainForm::MCPServerMCPRequestTool(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession,
    const TsgcAI_MCP_Request_ToolsCall *ARequest,
    const TsgcAI_MCP_Response_ToolsCall *AResponse)
{
    double LA, LB;
    if (ARequest->Params->Name == "math.add")
    {
        LA = ARequest->Params->Arguments->Node["a"]->AsNumber;
        LB = ARequest->Params->Arguments->Node["b"]->AsNumber;
        AResponse->Result->Content->AddText(
            Format("Sum = %.2f", ARRAYOFCONST((LA + LB)))
        );
    }
    else
    {
        AResponse->Result->IsError = true;
    }
}
```

生成的 JSON-RPC 响应将遵循规范的 tools.call 模式，包含一个或多个内容部分以及可选的结构化有效载荷。由于组件会为您清除响应并设置 HTTP 代码，因此无需额外的配置。

公共 API 接口

几个辅助类提供了工具目录和事件响应辅助工具。以下列表汇总了生产代码中通常调用的方法。

TsgcAI_MCP_ToolsList

- **AddTool(const Name, Description)**：创建或替换工具条目，并返回底层的 **TsgcAI_MCP_Tool**，以便配置架构和默认值。
- **Clear**：移除所有已注册的工具。在从磁盘加载配置后重建目录时调用此方法。
- **Count / Items[Index]**：公开列表内容以供枚举、诊断或自定义序列化。

TsgcAI_MCP_Tool

- **InputSchema.Properties.AddProperty(Name, Required, JsonType, Description)**：记录将通过 tools.list 公告的 JSON 输入载荷。
- **InputSchema.Properties.Clear**：当您的应用程序在运行时刷新工具能力时，动态重建 schema。
- **Description**：可写属性，存储与客户端共享的人类可读摘要。

TsgcAI_MCP_Response_ToolsCall

- **Result.Content.AddText(Value)**：将文本块追加到工具响应中。多次调用产生多部分响应。
- **Result.Content.AddImage(Data, MimeType)**：将图像块附加到工具响应。多次调用会产生多部分响应。
- **Result.Content.AddAudio(Data, MimeType)**：向工具响应追加音频块。多次调用可生成多部分响应。
- **Result.Content.AddEmbeddedResource(Uri, Title, Text, MimeType)**：向工具响应追加嵌入资源块。多次调用将产生多部分响应。
- **Result.Content.AddResourceLink(Uri, Name, MimeType)**：将资源链接块追加到工具响应中。多次调用将产生多部分响应。
- **Result.Content.Clear**：在追加新内容之前重置任何之前生成的输出。
- **Result.IsError**：当工具调用无法完成时，将此标志设为 **True**，框架将序列化 MCP 错误对象。

高级示例

下一段代码构建了一个地理位置查询工具，该工具验证输入，返回多个文本块，并在参数无效时将 **IsError** 翻转为 **True**。

```

void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    TsgcAI_MCP_Tool *tool = MCPServer->Tools->AddTool("geo.locate", "Returns coordinates for a city and country")
    tool->InputSchema->Properties->Clear();
    tool->InputSchema->Properties->AddProperty("city", true, aimcpjtString, "City to search");
    tool->InputSchema->Properties->AddProperty("country", false, aimcpjtString, "Optional ISO country code");
}
void __fastcall TMainForm::MCPServerMCPRequestTool(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession,
    const TsgcAI_MCP_Request_ToolsCall *ARequest,
    const TsgcAI_MCP_Response_ToolsCall *AResponse)
{
    if (!SameText(ARequest->Params->Name, "geo.locate"))
        return;
    UnicodeString city = ARequest->Params->Arguments->Node["city"]->AsString();
    UnicodeString country = ARequest->Params->Arguments->Node["country"]->AsString();
    AResponse->Result->Content->Clear();
    if (city.IsEmpty())
    {
        AResponse->Result->Content->AddText("City cannot be empty.");
        AResponse->Result->IsError = true;
        return;
    }
    AResponse->Result->Content->AddText("City: " + city);
    if (!country.IsEmpty())
        AResponse->Result->Content->AddText("Country: " + country);
    AResponse->Result->Content->AddText("Coordinates: " + LookupCityCoordinates(city, country));
}

```

MCP 服务器 | 提示词

在 MCP 中，*提示*是服务器公开的可复用模板或工作流。它们由用户控制（客户端选择要调用的提示）。主要有两种方法：

- **prompts/list**：请求服务器暴露的所有可用提示词定义的列表。由 TsgcWSAPIServer_MCP 组件在内部处理。
- **prompts/get**：检索特定提示词的内容及可选元数据，通常是描述客户端如何交互的消息序列。当服务器收到新请求时，**OnMCPRequestPrompt** 事件被调用。

此外，若服务器在其声明的能力中将 `prompts.listChanged` 设置为 `true`，则在提示词目录更新时，它可能会发送 `notifications/prompts/list_changed` 之类的通知。

当客户端请求 `prompts/get` 时，会发送参数（如有）并期望收到一系列消息（用户或助手角色），其中包含内容块（通常为文本，但也可能嵌入资源或图像）。

如果出现问题（缺少提示、参数无效），服务器将返回 JSON-RPC 错误代码，例如 -32602（无效参数）或 -32603（内部错误）。

提示词列表

服务器在 **TsgcAI_MCP_PromptsList** 中维护一份内存中的提示目录。提示以名称为唯一键，公开描述并提供类似 JSON Schema 的输入描述，该描述在规范的 `prompts.list` 方法响应中输出。当客户端调用 `prompts.list` 时，TsgcWSAPIServer_MCP 加载请求，序列化当前目录并以 200 HTTP 状态码返回。

发布代码评审提示的示例代码：

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    TsgcAI_MCP_Prompt* oPrompt;
    // Clear all existing prompts
    MCPServer->Prompts->Clear();
    // Create a new prompt named "CodeReview"
    oPrompt = MCPServer->Prompts->AddPrompt(
        "CodeReview",
        "Asks the LLM to analyze code quality and suggest improvements"
    );
    // Add argument 'code'
    oPrompt->Arguments->AddArgument(
        "code",
        "The code to review",
        true // required
    );
}
```

提示词请求

当客户端发出 `prompts.get` JSON-RPC 请求时，**TsgcWSAPIServer_MCP** 会在触发 **OnMCPRequestPrompt** 事件之前，将强类型请求对象（包括提示名称和所提供的参数）注入其中。您的处理程序填充响应有效载荷，然后将其连同成功的 HTTP 状态码序列化并返回给客户端。

典型的处理程序如下所示：

```
void __fastcall TFRMMCPServer::MCPServerMCPRequestPrompt(
    TObject *Sender,
    const TsgcAI_MCP_Session *aSession,
    const TsgcAI_MCP_Request_PromptsGet *aRequest,
    const TsgcAI_MCP_Response_PromptsGet *aResponse)
{
    // ...
}
```



```

if (aRequest->Params->Name == "CodeReview")
{
    aResponse->Result->Description = "Code review prompt";
    aResponse->Result->Messages->AddText(
        "user",
        "Please review this Delphi code: ShowMessage('Hello World');");
};
}
}

```

公共 API 接口

提示词通过少量辅助类建模。以下方法在整理提示词模板和组合 **prompts/get** 响应时最为常用。

TsgcAI_MCP_PromptsList

- **AddPrompt(const Name, Description)**：注册一个新的提示条目，并返回一个 **TsgcAI_MCP_Prompt** 对象，以便配置参数和消息。
- **Clear**：清空目录，在从配置文件加载提示词的启动阶段非常方便。
- **Count / Items[Index]**：提供对列表内容的访问，用于诊断、UI 可视化或手动序列化。

TsgcAI_MCP_Prompt

- **Arguments.AddArgument(Name, Description, Required)**：声明提示词接受的结构化输入。
- **Arguments.Clear**：在提示在运行时更改时重建参数列表。
- **Messages.AddText(Role, Text)**：预填充对话轮次，作为提示定义的一部分发送回客户端。

TsgcAI_MCP_Response_PromptsGet

- **Result.Description**：随提示载荷附带的简短可读摘要。
- **Result.Messages.AddText(Role, Text)**：在获取提示词时动态注入额外消息。
- **Result.Messages.AddImage(Data, MimeType)**：向提示词响应中追加图像块。多次调用会生成多部分响应。
- **Result.Messages.AddAudio(Data, MimeType)**：将音频块追加到提示响应中。多次调用可产生多部分响应。
- **Result.Messages.AddEmbeddedResource(Uri, Title, Text, MimeType)**：向提示响应追加嵌入资源块。多次调用可生成多部分响应。
- **Result.Messages.Clear**：在复用对象之前重置响应。

高级示例

下一个示例发布一个引导 LLM 完成分诊工作流的提示。它重点介绍了可选参数以及如何在处理程序中返回多步骤指令。

```

void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    TsgcAI_MCP_Prompt *prompt;
    MCPServer->Prompts->Clear();
    prompt = MCPServer->Prompts->AddPrompt("IssueTriage", "Collect details before escalating support tickets");
    prompt->Arguments->Clear();
    prompt->Arguments->AddArgument("summary", "One-line description supplied by the user", true);
    prompt->Arguments->AddArgument("customerPriority", "Optional priority label", false);
    prompt->Messages->AddText("system", "You are a support assistant that triages technical issues.");
}

void __fastcall TMainForm::MCPServerMCPRequestPrompt(
    TObject *Sender,
    const TsgcAI_MCP_Session *aSession,

```

```
const TsgcAI_MCP_Request_PromptsGet *aRequest,
const TsgcAI_MCP_Response_PromptsGet *aResponse)
{
    if (!SameText(aRequest->Params->Name, "IssueTriage"))
        return;
    aResponse->Result->Description = "Guided checklist for support analysts.";
    aResponse->Result->Messages->Clear();
    aResponse->Result->Messages->AddText("user", "Review the ticket summary and ask clarifying questions.");
    aResponse->Result->Messages->AddText("assistant", "Acknowledge the request and confirm the escalation path.")
    if (aRequest->Params->Arguments->Node["customerPriority"]->AsString() != "")
        aResponse->Result->Messages->AddText("assistant", "Adjust SLA checks based on the provided customer prior
}
```

MCP 服务器 | 资源

在 MCP 中，**资源**代表服务器公开的可寻址数据对象，例如文件、数据库记录、生成的文档或动态 API 输出。

它们是**客户端控制的**，即客户端可以决定请求哪个资源以及如何解释内容。

涉及的两个主要方法为：

- **resources/list**：请求列出服务器公开的所有可用资源定义。该请求由 `TsgcWSAPIServer_MCP` 组件在内部处理。
- **resources/read**：检索特定资源的内容及可选元数据。当服务器收到新请求时，将调用事件 **OnMCPRequestResource**。资源通常包含：
 - **uri**：资源标识符。
 - **mimeType**：内容类型（text/plain、application/json、image/png 等）。
 - **data 或 content**：实际负载（文本或二进制）。

此外，如果服务器在其声明的能力中设置了 `resources.listChanged = true`，则当资源目录更新时，它可能会发送 `notifications/resources/list_changed` 等通知。

资源列表

服务器在 `TsgcAI_MCP_ResourcesList` 中维护一个内存资源目录。资源通过 uri 保证唯一性，提供描述，并提供类似 JSON Schema 的输入 URI，该 URI 在响应规范的 `resources.list` 方法时被发送出去。当客户端调用 `resources.list` 时，`TsgcWSAPIServer_MCP` 加载请求，序列化当前目录，并以 200 HTTP 状态码返回。

发布文件资源的示例代码：

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    // Clear any previously registered resources
    MCPServer->Resources->Clear();
    // Register a Rust source file resource
    MCPServer->Resources->AddResource(
        "file:///project/src/main.rs",    // URI
        "main.rs",                        // Name
        "Rust Software Application Main File", // Title
        "Primary application entry point", // Description
        "text/x-rust"                     // MIME type
    );
}
```

资源请求

当客户端发出 `resource.read` JSON-RPC 请求时，`TsgcWSAPIServer_MCP` 在触发 **OnMCPRequestResource** 事件之前，会将强类型请求对象（包括 URI 资源、名称和提供的参数）水化。您的处理程序填充响应有效载荷，然后该有效载荷被序列化回客户端，并附带成功的 HTTP 状态码。

典型的处理程序如下所示：

```
void __fastcall TMainForm::OnMCPRequestResource(
    TObject *Sender,
    const TsgcAI_MCP_Session *aSession,
    const TsgcAI_MCP_Request_ResourcesRead *aRequest,
    const TsgcAI_MCP_Response_ResourcesRead *aResponse)
{
    if (aRequest->Params->Uri == "file:///project/src/main.rs")
    {
        aResponse->Result->Contents->AddContentText(
            "file:///project/src/main.rs", // URI
        );
    }
}
```

```

        "main.rs", // Name
        "Rust Software Application Main File", // Description
        "text/x-rust", // MIME type
        "fn main() {\n    println!(\"Hello world!\");\n}" // Content
    );
}
}

```

公共 API 接口

Resources 结合了目录条目（**TsgcAI_MCP_Resource**）和 **OnMCPRequestResource** 内部使用的流式辅助方法。以下是您最常用到的方法。

TsgcAI_MCP_ResourcesList

- **AddResource(Uri, Name, Title, Description, MimeType)**：注册一个静态或动态生成的资源。
- **Clear**：移除所有资源定义，在从配置文件重新构建列表之前非常有用。
- **Count / Items[Index]**：公开目录以供枚举或诊断视图使用。

TsgcAI_MCP_Resource

- **MimeType / Title / Description**：描述返回给客户端的资源负载的可写属性。
- **Name / Uri**：使用这些属性可以在不重新注册资源条目的情况下更新标识符。

TsgcAI_MCP_Response_ResourcesRead

- **Result.Contents.AddContentText(Uri, Name, Title, MimeType, Text)**：向调用方发回文本负载。
- **Result.Contents.AddContentBlob(Uri, Name, Title, MimeType, Text)**：向调用方发送文本有效载荷。
- **Result.Contents.Clear**：在重用响应对象之前重置输出载荷。
- **Result.IsError**：当无法生成资源时，将响应标记为错误。

高级示例

以下示例发布一个返回聚合指标的 JSON 资源。它演示了如何验证传入的 URI 并根据请求制作不同的响应。

```

void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    MCPServer->Resources->Clear();
    MCPServer->Resources->AddResource(
        "metrics://daily",
        "DailyMetrics",
        "Daily KPI snapshot",
        "Summaries generated each night",
        "application/json");
}

void __fastcall TMainForm::MCPServerMCPRequestResource(
    TObject *Sender,
    const TsgcAI_MCP_Session *aSession,
    const TsgcAI_MCP_Request_ResourcesRead *aRequest,
    const TsgcAI_MCP_Response_ResourcesRead *aResponse)
{
    UnicodeString targetUri = aRequest->Params->Uri.LowerCase();
    aResponse->Result->Contents->Clear();
    if (targetUri == "metrics://daily")
    {
        aResponse->Result->Contents->AddContentText(
            "metrics://daily",
            "DailyMetrics",
            "Daily KPI snapshot",
            "application/json",
            BuildMetricsPayload(Date()));
    }
}

```

```
        return;
    }
    aResponse->Result->IsError = true;
    aResponse->Result->Contents->AddContentText(
        aRequest->Params->Uri,
        "UnknownResource",
        "Not found",
        "text/plain",
        "Resource is not published by this server.");
}
```

MCP 服务器 | Roots

根目录标记客户端允许服务器浏览的文件系统位置。每个条目都是带有可选显示名称的 file:// URI，让 MCP 自动化遵守项目边界，同时用户保持对共享内容的控制。服务器通过发出 roots/list 查询目录，当客户端宣告 listChanged 时，接收 notifications/roots/list_changed 以刷新缓存视图。

由于根由客户端拥有，服务器始终充当请求者。在尝试发现调用之前，客户端的初始化有效载荷中必须包含 roots 能力；否则 RequestRootsList 会抛出 CapabilityNotSupported。

请求根目录

TsgcWSAPIServer_MCP 公开一个帮助器，将 roots/list 序列化并将回复路由回您的代码：

- **请求辅助工具**：一旦 MCP 会话宣告 roots 功能，即可调用 RequestRootsList(Session)。该方法会自动分配标识符（除非您提供了一个），并保留待处理请求，以便将响应进行关联。
- **响应事件**：处理 OnMCPResponseRootsList(Sender, Session, Request, Response)。当客户端回复可用目录时触发该事件。检查 Response.Roots 或记录 Response.Write 以检查 JSON 载荷。
- **变更检测**：若客户端随后发送 notifications/roots/list_changed，请再次提交 RequestRootsList 以保持服务器端缓存同步。

示例代码

下方的代码片段在会话建立后立即请求根目录，并记录返回的条目。

```
void __fastcall TFRMMCPServer::FormCreate(TObject *Sender)
{
    MCPServer->OnMCPSessionNew = MCPServerSessionNew;
    MCPServer->OnMCPResponseRootsList = MCPServerRootsList;
}

void __fastcall TFRMMCPServer::MCPServerSessionNew(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession)
{
    if (ASession->ClientCapabilities->Roots->Enabled)
    {
        System::UnicodeString requestId = MCPServer->RequestRootsList(ASession);
        MemoLog->Lines->Add("roots/list sent id " + requestId);
    }
}

void __fastcall TFRMMCPServer::MCPServerRootsList(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession,
    const TsgcAI_MCP_Request_RootsList *ARequest,
    const TsgcAI_MCP_Response_RootsList *AResponse)
{
    MemoLog->Lines->Add(
        Format("roots/list returned %d entries", ARRAYOFCONST((AResponse->Roots->Count)))
    );
    MemoLog->Lines->Add("roots/list payload " + AResponse->Write());
}
```

MCP 服务器 | 采样

采样让服务器代表客户端请求运行 LLM 生成。请求描述了对话历史、可选的系统提示、模型偏好和令牌预算。客户端保留对使用哪个提供商的控制权，执行任何人工审查，最后返回已批准的补全结果。

客户端必须在 initialize 期间声明 sampling 能力；否则采样尝试将被拒绝并显示能力错误。规范鼓励用户审批，并允许在对话历史中包含文本、图像或音频内容块。

发起采样请求

TsgcWSAPIServer_MCP 通过一个辅助方法和一个响应事件简化了 sampling/createMessage 流程：

- **请求辅助工具**：构建一个 **TsgcAI_MCP_Request_SamplingCreateMessage**，填充 **Params.Messages**、**Params.ModelPreferences**、**Params.SystemPrompt** 以及可选的 **Params.MaxTokens**。将其传递给 **RequestSamplingCreateMessage(Session, Request)**；该方法将在需要时生成 id 并记录待处理的调用。
- **响应事件**：处理 **OnMCPResponseSamplingCreateMessage(Sender, Session, Request, Response)**。检查 **Response.Role**、**Response.Content**、**Response.Model** 和 **Response.StopReason** 以继续您的工作流程。
- **错误处理**：被拒绝或取消的请求以 JSON-RPC 错误形式到达，因此请将辅助工具调用包裹在 try/except 中，并监控 **OnMCPEXception** 以进行日志记录。

示例代码

以下代码片段向客户端请求一段简短摘要并打印助手的回复。

```
void __fastcall TFRMMCPServer::FormCreate(TObject *Sender)
{
    MCPServer->OnMCPResponseSamplingCreateMessage = MCPServerSamplingResponse;
}
void __fastcall TFRMMCPServer::RequestTicketSummary(
    const TsgcAI_MCP_Session *ASession,
    const System::UnicodeString ATicketText)
{
    if (!ASession->ClientCapabilities->Sampling->Enabled)
        return;
    std::unique_ptr<TsgcAI_MCP_Request_SamplingCreateMessage> request(
        new TsgcAI_MCP_Request_SamplingCreateMessage());
    request->Params->Messages->AddTextMessage(
        "user",
        "Summarise the following ticket: " + ATicketText
    );
    request->Params->SystemPrompt = "You are a concise support assistant.";
    request->Params->ModelPreferences->Hints->AddHint("claude-3-sonnet");
    request->Params->ModelPreferences->SpeedPriority = 0.6;
    request->Params->ModelPreferences->IntelligencePriority = 0.8;
    request->Params->HasMaxTokens = true;
    request->Params->MaxTokens = 200;
    MemoLog->Lines->Add(
        "sampling/createMessage id " +
        MCPServer->RequestSamplingCreateMessage(ASession, request.get())
    );
}
void __fastcall TFRMMCPServer::MCPServerSamplingResponse(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession,
    const TsgcAI_MCP_Request_SamplingCreateMessage *ARequest,
    const TsgcAI_MCP_Response_SamplingCreateMessage *AResponse)
{
    if (dynamic_cast<TsgcAI_MCP_Response_Result_Content_Text*>(AResponse->Content) != nullptr)
    {
        auto *text = static_cast<TsgcAI_MCP_Response_Result_Content_Text*>(AResponse->Content);
        MemoLog->Lines->Add("assistant: " + text->Text);
    }
    MemoLog->Lines->Add(
        "model: " + AResponse->Model + " reason: " + AResponse->StopReason
    );
}
```

MCP Server | Elicitation

Elicitation 允许服务器在工作流中间要求客户端从用户处收集结构化输入。请求包含一条人性化消息以及描述必须捕获的字段的受限 JSON 模式。客户端渲染适当的表单，在本地验证答案，并返回用户是接受、拒绝还是取消了提示。

客户端必须在 initialize 期间声明 elicitation 能力。模式被有意限制为具有原始属性（字符串、数字、布尔值或枚举）的扁平对象，以简化渲染并将敏感数据排除在范围之外。

请求用户输入

TsgcWSAPIServer_MCP 为 elicitation/create 提供辅助，并在用户完成交互时引发响应事件：

- **请求辅助方法**：创建 `TsgcAI_MCP_Request_ElicitationCreate`，设置 `Params_Message`，并通过 `Params.RequestedSchema` 描述预期的有效载荷。调用 `RequestElicitationCreate(Session, Request)` 将其发送给客户端。
- **响应事件**：处理 `OnMCPResponseElicitationCreate(Sender, Session, Request, Response)`。检查 `Response.Action` (accept、decline 或 cancel)，并在用户接受时解析 `Response.Content`。
- **验证**：被拒绝的请求以 JSON-RPC 错误的形式到达。在使用任何内容之前，请始终检查 `Action`，并准备好在用户取消时重新请求。

示例代码

以下代码片段请求联系人信息，等待回复，并记录生成的操作及其负载。

```
void __fastcall TFRMMCPServer::FormCreate(TObject *Sender)
{
    MCPServer->OnMCPResponseElicitationCreate = MCPServerElicitationResponse;
}
void __fastcall TFRMMCPServer::RequestContactDetails(
    const TsgcAI_MCP_Session *ASession)
{
    if (!ASession->ClientCapabilities->Elicitation->Enabled)
        return;
    std::unique_ptr <TsgcAI_MCP_Request_ElicitationCreate> request(
        new TsgcAI_MCP_Request_ElicitationCreate());
    request->Params->Message = "Please confirm your contact information.";
    request->Params->RequestedSchema->Title = "Contact information";
    request->Params->RequestedSchema->Description =
        "Used to keep you updated about this ticket.";
    auto *name = request->Params->RequestedSchema->Properties->AddProperty("name");
    name->_Type = "string";
    name->Title = "Full name";
    name->Required = true;
    auto *email = request->Params->RequestedSchema->Properties->AddProperty("email");
    email->_Type = "string";
    email->Title = "Email address";
    email->Format = "email";
    email->Required = true;
    auto *updates = request->Params->RequestedSchema->Properties->AddProperty("updates");
    updates->_Type = "boolean";
    updates->Title = "Receive status updates";
    updates->Description = "Tick to receive notifications when the ticket changes.";
    updates->DefaultBoolean = false;
    MemoLog->Lines->Add(
        "elicitation/create id " +
        MCPServer->RequestElicitationCreate(ASession, request.get())
    );
}
void __fastcall TFRMMCPServer::MCPServerElicitationResponse(
    TObject *Sender,
    const TsgcAI_MCP_Session *ASession,
    const TsgcAI_MCP_Request_ElicitationCreate *ARequest,
    const TsgcAI_MCP_Response_ElicitationCreate *AResponse)
{
    MemoLog->Lines->Add("elicitation action: " + AResponse->Action);
    if (AResponse->Action.LowerCase() == "accept" && AResponse->Content != nullptr)
        MemoLog->Lines->Add("payload: " + AResponse->Content->Text);
}
```


TsgcWSAPIClient_MCP

TsgcWSAPIClient_MCP 组件。

简介

TsgcWSAPIClient_MCP 在 `sgcWebSockets` 网络组件之上实现了模型上下文协议（MCP）消费者。它处理会话协商、JSON-RPC 请求/响应编组以及便捷事件，使 Delphi 和 Cbuilder 应用程序能够发现任何符合 MCP 规范的服务器所公开的提示词、资源和工具。

配置

- **拖放组件**：将 **TsgcWSAPIClient_MCP** 放置到窗体或数据模块上。
- **指向 MCP 端点**：将 **MCPOptions.HttpOptions.URL** 设置为公开该协议的服务器 HTTP URL（例如 `https://localhost:5001/mcp`）。如有需要，通过嵌套的 **TLSOptions** 和 **LogOptions** 对象配置 TLS 和日志记录。
- **描述客户端**：在 **MCPOptions.ClientInfo** 中填充产品 *Name*、人类可读的 *Title* 和语义 *Version*。这些值在 **initialize** 握手期间被转发。
- **保护会话**：当服务器需要 API 密钥或自定义 HTTP 标头时，配置 **MCPOptions.AuthenticationOptions**。凭证将附加到每个 HTTP 请求，包括流式升级请求。
- **调整保活**：使用 **MCPOptions.HeartBeat** 启用自动 ping 周期并定义首选间隔（秒）。当内部保活计时器触发时，组件会在内部调用 **Ping**。
- **处理事件**：订阅后面描述的事件，以便记录响应、接受/拒绝会话或对服务器返回的负载进行后处理。

参考

- 属性 — 组件上有 2 个可用属性。
- 方法 — 12 个公共方法。
- 事件 — 12 个可处理的事件。

MCP 客户端 | 工具

TsgcWSAPIClient_MCP 组件公开了模型上下文协议（MCP）定义的 JSON-RPC 调用，用于枚举服务器提供的工具并调用工具。每个请求必须在 Initialize 成功后发送，因为每个方法都会验证会话是否已处于 *initialized* 状态。

ListTools 请求

调用 **ListTools** 以获取 MCP 服务器当前公开的工具注册表。组件会自动构建 tools/list JSON-RPC 请求，并以解析后的响应触发 **OnMCPListTools** 事件。

- **请求**：ListTools（无参数）。
- **事件**：OnMCPListTools(Sender, Request, Response)。
- **Response**：TsgcAI_MCP_Response_ToolsList，包含一个 Tools 队列。每个条目是一个 TsgcAI_MCP_Tool，公开工具的 Name、Title、Description 及其 InputSchema（JSON 模式类型加命名属性）。

使用队列的 Count 和 Item[Index] 辅助方法或 GetTool 来检查目录。

ToolsCall 请求

调用 **RequestTool**(Name, Arguments) 以调用其中一个已发布的工具。组件将 JSON 参数复制到请求负载中，并在解析响应后触发 **OnMCPResponseTool**。

- **请求**：RequestTool，包含工具名称和可选的 JSON 参数。
- **事件**：OnMCPResponseTool(Sender, Request, Response)。
- **响应**：TsgcAI_MCP_Response_ToolsCall。检查 Result.IsError、Result.StructuredContent，并遍历 Result.Content 获取结构化输出片段。工具响应可包含文本、图像、音频二进制数据、嵌入资源或资源链接。

示例代码

以下代码片段展示了如何请求工具清单、执行工具并读取结果。

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPListTools = MCPListTools;
    MCP->OnMCPResponseTool = MCPResponseTool;
}
void __fastcall TForm1::LoadTools()
{
    MCP->ListTools();
}
void __fastcall TForm1::MCPListTools(TObject *Sender,
const TsgcAI_MCP_Request_ToolsList &ARequest,
const TsgcAI_MCP_Response_ToolsList &AResponse)
{
    for (int LIndex = 0; LIndex < AResponse.Tools->Count; LIndex++)
    {
        TsgcAI_MCP_Tool *LTool =
            static_cast<TsgcAI_MCP_Tool*>(AResponse.Tools->Item[LIndex]);
        Memo1->Lines->Add(Format("%s (%s)", ARRAYOFCONST((LTool->Name, LTool->Description))));
    }
}
void __fastcall TForm1::CallTemperatureTool()
{
    TsgcJSON *LArgs = new TsgcJSON(nullptr);
    try
    {
        LArgs->AddPair("city", "Barcelona");
        MCP->RequestTool("GetTemperature", LArgs);
    }
    finally
    {
        delete LArgs;
    }
}
void __fastcall TForm1::MCPResponseTool(TObject *Sender,
const TsgcAI_MCP_Request_ToolsCall &ARequest,
const TsgcAI_MCP_Response_ToolsCall &AResponse)
```

```
{
    if (AResponse.Result->IsError)
        Memo1->Lines->Add("Tool returned an error payload");
    for (int LIndex = 0; LIndex < AResponse.Result->Content->Count; LIndex++)
    {
        TsgcAI_MCP_Response_Result_Content *LContent =
            static_cast<TsgcAI_MCP_Response_Result_Content*>(
                AResponse.Result->Content->Item[LIndex]);
        Memo1->Lines->Add(LContent->Write());
    }
}
```

MCP 客户端 | 提示

MCP 客户端组件提供两种围绕服务器提供的提示词进行操作的 JSON-RPC 方法：列出可用的提示词模板，以及请求执行提示词。请确保在发出这些调用之前已完成客户端初始化。

PromptsList 请求

ListPrompts 请求当前提示目录。收到响应时，组件触发 **OnMCPListPrompts**。

- **Request**：ListPrompts（无参数）。
- **事件**：OnMCPListPrompts(Sender, Request, Response)。
- **响应**：TsgcAI_MCP_Response_PromptsList，暴露一个 Prompts 队列。每个 TsgcAI_MCP_Prompt 包含其 Name、Title、Description 和 Arguments 集合（参数名称、描述和必填标志）。

PromptsGet 请求

RequestPrompt(Name, Arguments) 使用提供的 JSON 参数执行提示模板。服务器响应就绪后，组件触发 **OnMCPResponsePrompt**。

- **Request**：RequestPrompt，包含提示的 **Name** 和可选的 JSON **Arguments**。
- **事件**：OnMCPResponsePrompt(Sender, Request, Response)。
- **响应**：TsgcAI_MCP_Response_PromptsGet。检查 Result.Description 以获取摘要，并遍历 Result.Messages 读取每条 TsgcAI_MCP_Response_PromptsGet_Result_Message。消息具有多态性，请使用具体子类（..._Text、..._Image、..._Audio、..._EmbeddedResource）访问底层内容和注解。

示例代码

以下示例加载提示列表，调用一个提示，并打印返回的消息。

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPListPrompts = MCPListPrompts;
    MCP->OnMCPResponsePrompt = MCPResponsePrompt;
}
void __fastcall TForm1::LoadPrompts()
{
    MCP->ListPrompts();
}
void __fastcall TForm1::MCPListPrompts(TObject *Sender,
const TsgcAI_MCP_Request_PromptsList &ARequest,
const TsgcAI_MCP_Response_PromptsList &AResponse)
{
    for (int LIndex = 0; LIndex < AResponse.Prompts->Count; LIndex++)
    {
        TsgcAI_MCP_Prompt *LPrompt = (TsgcAI_MCP_Prompt*)AResponse.Prompts->Item[LIndex];
        Memo1->Lines->Add(Format("%s: %s", ARRAYOFCONST((LPrompt->Name, LPrompt->Description))));
    }
}
void __fastcall TForm1::ExecutePrompt()
{
    TsgcJSON *LArgs = new TsgcJSON(nullptr);
    try
    {
        LArgs->AddPair("code", "ShowMessage('Hello World')");
        MCP->RequestPrompt("CodeReview", LArgs);
    }
    __finally
    {
        delete LArgs;
    }
}
void __fastcall TForm1::MCPResponsePrompt(TObject *Sender,
const TsgcAI_MCP_Request_PromptsGet &ARequest,
const TsgcAI_MCP_Response_PromptsGet &AResponse)
{
    Memo1->Lines->Add("Prompt description: " + AResponse.Result->Description);
}
```

```
for (int LIndex = 0; LIndex < AResponse.Result->Messages->Count; LIndex++)
{
    TsgcAI_MCP_Response_PromptsGet_Result_Message *LMessage =
        (TsgcAI_MCP_Response_PromptsGet_Result_Message*)AResponse.Result->Messages->Item[LIndex];
    if (dynamic_cast<TsgcAI_MCP_Response_PromptsGet_Result_Message_Text*>(LMessage))
    {
        auto *MsgText = (TsgcAI_MCP_Response_PromptsGet_Result_Message_Text*)LMessage;
        Memo1->Lines->Add(LMessage->Role + ": " + MsgText->Content.Text);
    }
    else
    {
        Memo1->Lines->Add(LMessage->Role + ": " + LMessage->Write());
    }
}
```

MCP 客户端 | 资源

MCP 资源 API 允许客户端通过 **TsgcWSAPIClient_MCP** 组件在会话初始化后发现服务器端产物并获取其内容。

ResourcesList 请求

使用 **ListResources** 获取资源目录。组件在解析 JSON-RPC 回复后触发 **OnMCPListResources**。

- **请求**：ListResources（无参数）。
- **事件**：OnMCPListResources(Sender, Request, Response)。
- **响应**：TsgcAI_MCP_Response_ResourcesList，公开一个 Resources 队列。每个 TsgcAI_MCP_Resource 条目包含资源的 Uri、Name、Title、Description 和 MimeType。

ResourcesRead 请求

调用 **RequestResource(Uri)** 以下载特定资源。组件将触发 **OnMCPResponseResource**，其中包含已解码的载荷。

- **请求**：以目标 **Uri** 调用 RequestResource。
- **事件**：OnMCPResponseResource(Sender, Request, Response)。
- **响应**：TsgcAI_MCP_Response_ResourcesRead。读取 Result.Contents 以访问每个 TsgcAI_MCP_Response_Resources_ReadResult_Content 块，其中包含 Uri、Name、Title、MimeType，以及 Text 或 Blob 数据。

示例代码

此代码片段列出资源并检索单个文件。

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPListResources = MCPListResources;
    MCP->OnMCPResponseResource = MCPResponseResource;
}
void __fastcall TForm1::LoadResources()
{
    MCP->ListResources();
}
void __fastcall TForm1::MCPListResources(TObject *Sender,
const TsgcAI_MCP_Request_ResourcesList &ARequest,
const TsgcAI_MCP_Response_ResourcesList &AResponse)
{
    for (int LIndex = 0; LIndex < AResponse.Resources->Count; LIndex++)
    {
        TsgcAI_MCP_Resource *LResource =
            static_cast<TsgcAI_MCP_Resource*>(AResponse.Resources->Item[LIndex]);
        Memo1->Lines->Add(Format("%S -> %S", ARRAYOFCONST((LResource->Name, LResource->Uri))));
    }
}
void __fastcall TForm1::FetchResource(const String AUri)
{
    MCP->RequestResource(AUri);
}
void __fastcall TForm1::MCPResponseResource(TObject *Sender,
const TsgcAI_MCP_Request_ResourcesRead &ARequest,
const TsgcAI_MCP_Response_ResourcesRead &AResponse)
{
    for (int LIndex = 0; LIndex < AResponse.Result->Contents->Count; LIndex++)
    {
        TsgcAI_MCP_Response_Resources_ReadResult_Content *LContent =
            static_cast<TsgcAI_MCP_Response_Resources_ReadResult_Content*>(
                AResponse.Result->Contents->Item[LIndex]);
        if (LContent->Text != "")
            Memo1->Lines->Add(LContent->Uri + ": " + LContent->Text);
        else
            Memo1->Lines->Add(LContent->Uri + " (binary, mime=" + LContent->MimeType + ")");
    }
}
```

```
}
```

MCP 客户端 | 根目录

MCP 规范引入了 **roots** 能力，使客户端能够描述其愿意向服务器公开的文件系统位置。每个 root 是一个 `file://` URI，附带可选的显示名称。服务器通过 `roots/list` 查询目录，并在支持的情况下，每当选择发生变化时收到 `notifications/roots/list_changed` 消息。此握手让工具能够遵守项目边界、工作区和多仓库布局，同时让用户控制服务器可以访问的内容。

理解 roots

根 (Roots) 作为范围服务器端功能的护栏。客户端在 `initialize` 期间通过启用 `roots` 能力（可选地启用 `listChanged` 标志以推送动态更新）来宣告支持。从 `roots/list` 返回的典型载荷包含一个或多个条目，每个条目具有 `Uri` 和 `Name`，代表工作区、存储库或其他精选文件夹。

处理 roots/list 请求

TsgcWSAPIClient_MCP 解析传入的 JSON-RPC 消息，并在服务器请求当前目录时触发 **OnMCPListRoots**。使用该事件将您的应用程序维护的任何工作区模型与协议响应同步。

- **请求：** `TsgcAI_MCP_Request_RootsList`。该消息不包含参数，但提供了 JSON-RPC 标识符，您必须在响应中镜像该标识符。
- **事件：** `OnMCPListRoots(Sender, Request, Response)`。此事件在组件验证会话后触发。
- **响应：** `TsgcAI_MCP_Response_RootsList`。调用 `Roots.AddRoot(Uri, Name)`（或手动构建 `TsgcAI_MCP_Root` 实例）来描述每个可访问的位置。组件在您的处理器返回后序列化填充好的对象，因此请确保列表反映当前状态。使用 `Response.Write` 进行诊断或日志记录。

当您的应用程序允许用户在运行时添加或移除工作区时，请记得触发相应的 `notifications/roots/list_changed` 通知，以便服务器刷新其缓存的目录。

示例代码

以下代码段展示了当服务器请求时如何发布两个根，并记录将要发回的 JSON 有效负载。

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPListRoots = MCPListRoots;
}
void __fastcall TForm1::MCPListRoots(TObject *Sender,
    const TsgcAI_MCP_Request_RootsList *ARequest,
    const TsgcAI_MCP_Response_RootsList *AResponse)
{
    AResponse->Roots->Clear();
    AResponse->Roots->AddRoot("file:///c:/projects/app", "Application Workspace");
    AResponse->Roots->AddRoot("file:///c:/projects/libs", "Shared Libraries");
    MemoLog->Lines->Add("Roots response: " + AResponse->Write());
}
```


MCP 客户端 | 采样

采样让 MCP 服务器可以在不共享提供商凭据的情况下，将语言模型的生成任务委托给客户端。服务器发出 `sampling/createMessage` 请求，该请求封装了对话状态、模型偏好以及 `maxTokens` 等可选约束。客户端可以引入人工审核员，执行防护措施，并最终返回包含所选模型、生成内容和停止原因的响应。

理解采样

当在 `initialize` 期间公告 `sampling` 能力时，服务器可以在工作流中途请求生成内容。请求捆绑消息数组（文本、图像或音频块）、`systemPrompt`、可选的令牌限制，以及通过数字优先级和提示字符串平衡成本、速度和智能的 `ModelPreferences` 结构。客户端负责将这些提示映射到可用模型、必要时提示用户，并在回复前过滤不安全的补全内容。

处理 `sampling/createMessage` 请求

`TsgcWSAPIClient_MCP` 通过 `OnMCPSamplingCreateMessage` 呈现服务器发起的生成请求。您的处理程序应检查提供的上下文，将调用路由到所需模型（或人工审核员），并填充响应负载。

- **请求**： `TsgcAI_MCP_Request_SamplingCreateMessage`。检查 `Params.Messages` 以重播对话，检查 `Params.ModelPreferences` 以了解提示优先级，以及 `Params.SystemPrompt` 以了解安全护栏。
- **事件**： `OnMCPSamplingCreateMessage(Sender, Request, Response)`。在组件验证 MCP 会话并反序列化请求后触发。
- **Response**： `TsgcAI_MCP_Response_SamplingCreateMessage`。提供助手 `Role`，分配内容对象（文本、图像、音频或资源链接），并可选记录具体的 `Model` 和 `StopReason`。调用 `Response.Write` 可在 JSON 负载分发前预览。

示例代码

以下代码片段记录传入的消息数量，生成固定回复，并标注模型和停止原因。在生产环境中，请将硬编码的内容替换为真实的模型调用或人工审核流程。

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPSamplingCreateMessage = MCPSamplingCreateMessage;
}
void __fastcall TForm1::MCPSamplingCreateMessage(TObject *Sender,
const TsgcAI_MCP_Request_SamplingCreateMessage *ARequest,
const TsgcAI_MCP_Response_SamplingCreateMessage *AResponse)
{
    MemoLog->Lines->Add(Format("Sampling request with %d messages", ARRAYOFCONST((ARequest->Params->Messages->Count))), TColorRed);
    AResponse->Role = "assistant";
    auto *text = new TsgcAI_MCP_Response_Result_Content_Text();
    text->Text = "The capital of France is Paris.";
    AResponse->Content = text;
    AResponse->Model = "claude-3-sonnet-20240307";
    AResponse->StopReason = "endTurn";
    MemoLog->Lines->Add("Sampling response: " + AResponse->Write());
}
```

MCP 客户端 | 引导

Elicitation 使服务器能够通过 MCP 客户端向用户请求结构化输入。请求携带一条可读消息和一个约束 JSON Schema，用于定义预期字段。客户端控制交互过程，验证提供的数据，并返回一个动作，指示用户是接受、拒绝还是取消了提示。

理解引导

elicitation 能力在 initialize 期间协商。一旦启用，服务器可发送 elicitation/create 调用，其中嵌入一个平面对象模式，仅限原始类型（string、number/integer、boolean 以及带可选显示名称的 enum）。每个属性可指定显示元数据、长度或范围约束及支持的格式（email、uri、date、date-time）。用户应始终被告知谁在请求信息，并有机会编辑负载，且可以拒绝或取消整个请求。

处理 elicitation/create 请求

每当服务器需要额外输入时，`TsgcWSAPIClient_MCP` 会触发 `OnMCPElicitationCreate`。该事件提供提示消息、请求的模式以及一个空的响应对象，您可以根据用户决策填充该对象。

- **请求：** `TsgcAI_MCP_Request_ElicitationCreate`。检查 `Params.Message` 以显示提示词，检查 `Params.RequestedSchema.Properties` 以构建 UI 控件或验证逻辑。
- **事件：** `OnMCPElicitationCreate(Sender, Request, Response)`。在客户端反序列化 JSON-RPC 有效载荷后触发。
- **Response：** `TsgcAI_MCP_Response_ElicitationCreate`。将 Action 设置为 `accept`、`decline` 或 `cancel`。接受时，将收集的值写入 `Content`（一个 `TsgcJSON` 辅助器），使其与请求的模式匹配。使用 `Response.Write` 检查最终 JSON 文档。

示例代码

以下示例提示用户输入 GitHub 用户名。如果操作员提供了值，则接受响应并返回字段；否则拒绝请求。

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    MCP->OnMCPElicitationCreate = MCPElicitationCreate;
}
void __fastcall TForm1::MCPElicitationCreate(TObject *Sender,
const TsgcAI_MCP_Request_ElicitationCreate *ARequest,
const TsgcAI_MCP_Response_ElicitationCreate *AResponse)
{
    MemoLog->Lines->Add("Elicitation prompt: " + ARequest->Params->Message);
    String gitHub = InputBox("GitHub", "Provide your GitHub username", "");
    if (!gitHub.IsEmpty())
    {
        AResponse->Action = "accept";
        AResponse->Content->Clear();
        AResponse->Content->AddPair("name", gitHub);
    }
    else
    {
        AResponse->Action = "decline";
    }
    MemoLog->Lines->Add("Elicitation response: " + AResponse->Write());
}
```

OpenAI

OpenAI 是一家私人研究实验室，旨在以造福全人类的方式开发和引导人工智能（AI）。OpenAI 开发了以下项目：

- **GPT-3**：此强大的语言模型是其他 OpenAI 产品的基础。它通过分析人类生成的文本来学习自主生成类似文本。
- **DALL-E 和 DALL-E 2**：这些生成式 AI 平台可以分析用户希望生成的图像的文字描述，然后完全按照描述生成这些图像。
- **CLIP**：CLIP 是一种神经网络，将视觉和相关文本合成，以预测最能准确描述这些视觉的字幕。由于其从多种数据类型（图像和文本）中学习的能力，它可以归类为多模态 AI。
- **ChatGPT**：ChatGPT 目前是最先进的 AI 聊天机器人，专为生成类人文本和回答用户问题而设计。经过大型数据集训练后，它能够像人类一样生成答案和响应。
- **Codex**：Codex 基于数十亿行多种编程语言的代码进行训练，旨在帮助软件开发人员简化编码流程。它建立在 GPT-3 技术之上，但不生成文本，而是生成代码。
- **Whisper**：Whisper 是一种自动语音识别（ASR）工具。它经过海量音频数据训练，能够识别、转录并翻译约 100 种语言的语音，包括技术语言 and 不同口音。

OpenAI API

OpenAI API 可应用于几乎所有涉及理解或生成自然语言、代码或图像的任务。OpenAI 提供具有不同能力级别的一系列模型，适用于不同任务，同时还提供对自定义模型进行微调的能力。这些模型可用于从内容生成到语义搜索和分类的各种场景。

最常见的用途

- 补全
 - [OpenAI 补全示例](#)
- 聊天
 - [OpenAI 聊天示例](#)
- 编辑
 - [OpenAI 编辑示例](#)
- 音频
 - [OpenAI 转录与翻译示例](#)
- 内容审核
 - [OpenAI 内容审核示例](#)
- RealTime
 - [OpenAI RealTime 示例](#)
- 响应
 - [OpenAI 响应示例](#)
- 语音
 - [OpenAI 语音示例](#)
- 微调
 - [OpenAI 微调示例](#)
- 批处理
 - [OpenAI 批处理示例](#)
- 上传
 - [OpenAI 上传示例](#)

配置

OpenAI

OpenAI API 使用 API 密钥进行身份验证。请访问您的 [API 密钥](#) 页面获取您在请求中使用的 API 密钥。请记住，您的 API 密钥是保密的！请勿与他人共享或在任何客户端代码（浏览器、应用程序）中公开。生产请求必须通过您自己的后端服务器路由，并从环境变量或密钥管理服务中安全加载 API 密钥。

此 **API Key** 必须在组件的 **OpenAIOptions.ApiKey** 属性中配置。对于属于多个组织的用户，如果您的账户属于某个组织，可以在 **OpenAIOptions.Organization** 属性中设置您的组织。

配置 API 密钥后，以下是与 OpenAI API 交互的可用函数列表。

Azure

该客户端支持 Microsoft Azure OpenAI Services，因此您也可以使用 Azure 账户与 Azure OpenAI API 进行交互。要将客户端配置为与 Azure 配合使用，请按照以下步骤操作：

1. 将属性 **OpenAIOptions.Provider** 配置为 `oapvAzure`
2. 设置 **ResourceName** 和 **DeploymentId** 的值（这些值可以在您的 Azure 账户中找到）
 1. **OpenAIOptions.AzureOptions.ResourceName** = <您的资源名称>。
 2. **OpenAIOptions.AzureOptions.DeploymentId** = <您的部署 ID>。
3. 设置您 Azure 账户的 API 密钥
 1. **OpenAIOptions.ApiKey** = <azure api key>。

请注意，并非所有 OpenAI 方法都受 Azure 支持，目前仅支持以下方法：

1. 完成
2. 聊天补全

属性

OpenAIOptions

- **ApiKey**：用于向 OpenAI API 进行身份验证的 API 密钥。
- **Organization**：属于多个组织的用户的可选组织 ID。
- **Provider**：选择提供商：`oapvOpenAI`（默认）或 `oapvAzure`（Microsoft Azure OpenAI 服务）。
- **AzureOptions**：Azure OpenAI 服务的配置。
 - **ResourceName**：Azure 资源名称。
 - **DeploymentId**：Azure 部署 ID。
 - **APIVersion**：Azure API 版本。
- **HttpOptions**：HTTP 连接设置。
 - **ReadTimeout**：读取 HTTP 响应的超时时间（毫秒）。默认值为 0（无超时）。
- **LogOptions**：日志配置。
 - **Enabled**：为 `True` 时，HTTP 请求和响应将被记录到文件中。
 - **FileName**：写入日志输出的文件路径。
- **RetryOptions**：失败 API 请求的自动重试配置。
 - **Enabled**：为 `True` 时，失败的请求将自动重试。
 - **Retries**：最大重试次数。
 - **Wait**：两次重试之间的等待时间（毫秒）。

模型

列出并描述 API 中可用的各种模型。

- **GetModels**：列出当前可用的模型，并提供每个模型的基本信息，例如所有者和可用性。
- **GetModel**：检索一个模型实例，提供关于该模型的基本信息，如所有者和权限设置。
 - **Model**：用于此请求的模型 ID。

补全

给定一个提示，模型将返回一个或多个预测的补全结果，还可返回每个位置上备选 token 的概率。

- **CreateCompletion**：为提供的提示词和参数创建补全内容。
 - **Model**：要使用的模型 ID。您可以使用列出模型 API 查看所有可用模型，或参阅模型概览了解其描述。
 - **Prompt**：用于生成补全内容的提示词。

聊天

给定聊天对话后，模型将返回聊天补全响应。

- **Model**：要使用的模型 ID。调用 GetModels 获取 Chat API 支持的所有模型列表。
- **Message**：用于生成聊天补全的消息。
- **Role**：默认为 user，其他选项包括：system、assistant。

编辑

给定提示词和指令，模型将返回提示词的编辑版本。

- **CreateEdit**：根据提供的输入、指令和参数创建新的编辑。
 - **Model**：要使用的模型 ID。可以在此端点使用 text-davinci-edit-001 或 code-davinci-edit-001 模型。
 - **Instruction**：告知模型如何编辑提示的指令。
 - **Input**：（可选）用作编辑起点的输入文本。

图像

给定提示词和/或输入图像，模型将生成新图像。

- **CreateImage**：根据提示词创建图像。
 - **Prompt**：所需图像的文本描述。最大长度为 1000 个字符。
- **CreateImageEdit**：根据原始图像和提示词创建编辑或扩展的图像。
 - **Image**：要编辑的图像。必须是有效的 PNG 文件，小于 4MB，且为正方形。如果未提供 mask，图像必须具有透明度，该透明度将用作蒙版。
 - **Prompt**：对所需图像的文本描述，最大长度为 1000 个字符。
- **CreateImageVariations**：基于给定图像创建变体。
 - **Image**：用于生成变体的基础图像。必须是有效的 PNG 文件，大小小于 4MB，且为正方形。

嵌入向量

获取给定输入的向量表示，该表示可供机器学习模型和算法轻松使用。

- **CreateEmbeddings**：创建表示输入文本的嵌入向量。
 - **Model**：要使用的模型 ID。
 - **Input**：用于获取嵌入的输入文本。

音频

将音频转换为文本。

- **CreateTranscriptionFromFile**：从文件名将音频转录为输入语言。
 - **Model**：要使用的模型 ID。目前仅支持 whisper-1。

- **Filename**：要转录的音频文件，支持以下格式之一：mp3、mp4、mpeg、mpga、m4a、wav 或 webm。
- **CreateTranscription**：录制 X 秒的音频并进行转录。
 - **Model**：要使用的模型 ID。目前只有 whisper-1 可用。
 - **Time**：时间（毫秒），默认为 10 秒。
- **CreateTranslationFromFile**：将音频翻译为英语。
 - **Model**：要使用的模型 ID。目前只有 whisper-1 可用。
 - **Filename**：要翻译的音频文件，格式为以下之一：mp3、mp4、mpeg、mpga、m4a、wav 或 webm。
- **CreateTranslation**：录制 X 秒音频并进行翻译。
 - **Model**：要使用的模型 ID。目前仅提供 whisper-1。
 - **Time**：时长（毫秒），默认 10 秒。

文件

文件用于上传可与微调等功能配合使用的文档。

- **ListFiles**：返回属于用户组织的文件列表。
- **UploadFile**：上传包含文档的文件，供各个端点/功能使用。目前，一个组织上传的所有文件总大小最多为 1 GB。
 - **Filename**：要上传的 JSON Lines 文件的名称。如果 purpose 设置为 "fine-tune"，则每行是一条带有 "prompt" 和 "completion" 字段的 JSON 记录，代表您的训练示例。
 - **Purpose**：上传文档的预期用途。对于微调，请使用 "fine-tune"。
- **DeleteFile**：删除文件。
 - **FileId**：用于此请求的文件 ID
- **RetrieveFile**：返回特定文件的信息。
 - **FileId**：本次请求所使用文件的 ID
- **RetrieveFileContent**：返回指定文件的内容
 - **FileId**：此请求使用的文件 ID。

Fine-Tunes

管理微调任务，根据您的特定训练数据定制模型。

- **CreateFineTune**：创建一个任务，使用给定的数据集对指定模型进行微调。响应包含已入队任务的详细信息，包括任务状态以及完成后微调模型的名称。
 - **TrainingFile**：包含训练数据的已上传文件的 ID。
- **ListFineTunes**：列出您的组织的微调任务。
- **RetrieveFineTune**：获取微调作业的信息。
 - **FineTuneId**：微调作业的 ID
- **CancelFineTune**：立即取消微调作业。
 - **FineTuneId**：微调作业的 ID
- **ListFineTuneEvents**：获取微调任务的细粒度状态更新。
 - **FineTuneId**：微调任务的 ID
- **DeleteFineTuneModel**：删除一个微调模型。您必须在组织中拥有 Owner 角色。
 - **Model**：要删除的模型。

内容审核

给定输入文本，输出模型是否将其归类为违反 OpenAI 内容政策。

- **CreateModeration**：分类文本是否违反 OpenAI 的内容政策
 - **Input**：要分类的输入文本

实时

OpenAI Realtime API 实现低延迟、多模态交互，包括语音到语音对话体验和实时转录。

- **转录**：您可以将 Realtime API 用于纯转录场景，输入可来自麦克风或文件。例如，您可以用它实时生成字幕或转录文本。在纯转录模式下，模型不会生成响应。

助手

构建可调用模型并使用工具执行任务的 AI 助手。

- **CreateAssistant**：使用模型和指令创建 Assistant。
- **ListAssistants**：返回助手列表。
- **RetrieveAssistant**：按 ID 检索助手。
- **ModifyAssistant**：修改现有助手。
- **DeleteAssistant**：删除一个助手。

线程

线程（Thread）代表一次对话会话。消息被添加到线程中，然后由运行（Run）进行处理。

- **CreateThread**：创建一个新线程。
- **RetrieveThread**：通过 ID 检索线程。
- **ModifyThread**：修改一个线程。
- **DeleteThread**：删除一个线程。

线程消息

消息被添加到线程中，并包含对话的内容。

- **CreateMessage**：在线程中创建一条消息。
 - **ThreadId**：线程的 ID。
- **ListMessages**：返回给定线程的消息列表。
- **RetrieveMessage**：通过线程和消息 ID 检索消息。
- **ModifyMessage**：修改消息。
- **DeleteMessage**：从线程中删除一条消息。

运行

Run 表示在线程上与 Assistant 一起执行的过程。Assistant 使用其配置和线程消息通过调用模型和工具来执行任务。

- **CreateRun**：为线程创建一个使用指定助手的运行。
- **ListRuns**：返回属于某个线程的 Run 列表。
- **RetrieveRun**：通过线程和运行 ID 检索运行。
- **ModifyRun**：修改运行。
- **SubmitToolOutputsToRun**：为需要操作的 Run 提交工具输出。
- **CancelRun**：取消正在进行的 Run。

运行步骤

运行步骤代表运行执行过程中采取的各个步骤。

- **ListRunSteps**：返回属于某次运行的运行步骤列表。
- **RetrieveRunSteps**：通过 thread、run 和 step ID 检索运行步骤。

向量存储

向量存储用于通过嵌入存储和搜索文件，以实现检索增强生成（RAG）。

- **CreateVectorStore**：创建一个向量存储。
- **ListVectorStores**：返回向量存储列表。
- **RetrieveVectorStore**：通过 ID 检索向量存储。
- **ModifyVectorStore**：修改向量存储。
- **DeleteVectorStore**：删除一个向量存储。

向量存储文件

管理向量存储中的文件。

- **CreateVectorStoreFile**：将文件附加到向量存储。
- **ListVectorStoreFiles**：返回向量存储中的文件列表。
- **RetrieveVectorStoreFile**：检索向量存储文件。
- **DeleteVectorStoreFile**：从向量存储中删除文件。

向量存储文件批次

向向量存储批量添加文件的操作。

- **CreateVectorStoreFileBatch**：创建一个批处理，将多个文件添加到向量存储。
- **RetrieveVectorStoreFileBatch**：按 ID 检索文件批次。
- **CancelVectorStoreFileBatch**：取消正在进行的文件批次。
- **ListVectorStoreFilesBatch**：列出批次中的文件。

语音合成

使用文字转语音模型从文本生成语音音频。

- **CreateSpeech**：根据输入文本生成音频。以字符串形式返回音频数据或写入响应流。
 - **Model**：要使用的 TTS 模型（例如 tts-1、tts-1-hd）。
 - **Input**：要生成音频的文本。
 - **Voice**：要使用的声音（alloy、echo、fable、onyx、nova、shimmer）。

微调任务

管理微调作业以创建自定义模型。这是新版微调 API，取代了旧版 Fine-Tunes 端点。

- **CreateFineTuningJob**：从训练文件创建微调任务。
- **ListFineTuningJobs**：列出您组织的微调任务。
- **RetrieveFineTuningJob**：检索有关微调作业的信息。
- **CancelFineTuningJob**：取消一个微调作业。
- **ListFineTuningJobEvents**：列出微调作业的状态更新。
- **ListFineTuningJobCheckpoints**：列出微调作业的检查点。

响应

创建和管理模型响应。Responses API 支持多轮对话、工具使用和结构化输出。

- **CreateResponse**：创建模型响应。
- **RetrieveResponse**：通过 ID 检索响应。
- **DeleteResponse**：删除响应。
- **CancelResponse**：取消正在进行的响应。
- **ListInputItems**：列出响应的输入项。

批处理

创建和管理批量 API 请求，以较低成本进行异步处理。

- **CreateBatch**：创建批处理作业。
- **RetrieveBatch**：通过 ID 检索批次。
- **ListBatches**：列出所有批处理任务。
- **CancelBatch**：取消正在进行中的批处理。

上传

分段上传大文件，适用于超过标准上传大小限制的文件。

- **CreateUpload**：创建上传会话。
- **AddUploadPart**：向上传任务添加一个分片。
- **CompleteUpload**：完成多部分上传并创建文件。
- **CancelUpload**：取消正在进行的上传。

TsgcHTTP_API_OpenAI

OpenAI REST API 客户端，涵盖聊天补全、图像、音频（TTS/STT）、嵌入、文件、微调、审核、助手、线程、运行、消息和向量存储。

简介

OpenAI REST API 客户端，涵盖聊天补全、图像、音频（TTS/STT）、嵌入、文件、微调、审核、助手、线程、运行、消息和向量存储。

简介

OpenAI REST API 客户端，涵盖聊天补全、图像、音频（TTS/STT）、嵌入、文件、微调、审核、助手、线程、运行、消息和向量存储。

使用您的 API 密钥及可选的 HTTP/重试/日志设置配置 **OpenAIOptions** 属性，然后调用类型化方法（如 `CreateMessage/CreateChatCompletion`），传入请求对象并接收类型化响应类。

有关该提供商的背景信息，请参阅 [OpenAI 指南](#)。

参考

- 属性 — 组件上提供 5 个属性。
- 方法 — 78 个公共方法。
- 事件 — 2 个可处理的事件。

OpenAI | 补全

给定一个提示词，模型将返回一个或多个预测的补全，还可以返回每个位置替代令牌的概率。

简单示例

使用 text-davinci-003 模型获取预测的补全结果。

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";
ShowMessage(OpenAI->_CreateCompletion("text-davinci-003", "Say this is a test"));
```

高级示例

使用 text-davinci-003 模型获取具有更多随机输出的预测补全，并为每个提示生成 2 个补全结果。

```
TsgcHTTP_OpenAI_JSON *OpenAI = new TsgcHTTP_OpenAI_JSON(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

TsgcOpenAIClass_Request_Completion *oRequest = new TsgcOpenAIClass_Request_Completion();
try
{
    oRequest->Model = "text-davinci-003";
    oRequest->Prompt = "Say this is a test";
    oRequest->Temperature = 1;
    oRequest->N = 2;
    TsgcOpenAIClass_Response_Completion *oResponse = OpenAI->CreateCompletion(oRequest);

    if (Length(oResponse->Choices) > 0)
    {
        ShowMessage(oResponse->Choices[0]->Text);
    }
}
__finally
{
    delete oRequest;
    delete oResponse;
}
```

OpenAI | Chat

给定聊天对话后，模型将返回聊天补全响应。

简单示例

通过发送 Hello 消息与 ChatGPT 进行交互。

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";
ShowMessage(OpenAI->_CreateChatCompletion("gpt-3.5-turbo", "Hello!"));
```

高级示例

使用 gpt-3-5 模型进行聊天，输出随机性更高，并为每个提示词生成 2 个补全结果。

```
TsgcHTTP_OpenAI_JSON *OpenAI = new TsgcHTTP_OpenAI_JSON(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

TsgcOpenAIClass_Request_ChatCompletion *oRequest = new TsgcOpenAIClass_Request_ChatCompletion();
try
{
    oRequest->Model = "gpt-3.5-turbo";
    TsgcOpenAIClass_Request_Completion_Message *oMessage = new TsgcOpenAIClass_Request_Completion_Message();
    oMessage->Role = "user";
    oMessage->Content = "Hello!";
    oMessages = oRequest->Messages;
    SetLength(oMessages, 1);
    oMessages[0] = oMessage;
    oRequest->Messages = oMessages;
    oRequest->Temperature = 1;
    oRequest->N = 2;
    TsgcOpenAIClass_Response_ChatCompletion *oResponse = OpenAI->CreateChatCompletion(oRequest);

    if (Length(oResponse->Choices) > 0)
    {
        ShowMessage(oResponse->Choices[0]->_Message->Content);
    }
}
__finally
{
    delete oRequest;
    delete oResponse;
}
```

OpenAI | 编辑

给定提示和指令，模型将返回提示的编辑版本。

简单示例

让 OpenAI 修正提示词中的拼写错误。

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";
ShowMessage(OpenAI->_CreateEdit("text-davinci-edit-001", "Fix the spelling mistakes", "What day of the wek is it?"
```

高级示例

告知 OpenAI 修正提示中的拼写错误，增加输出随机性，并为每个提示生成 2 个完成结果。

```
TsgcHTTP_OpenAI_JSON *OpenAI = new TsgcHTTP_OpenAI_JSON(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

TsgcOpenAIClass_Request_Edit *oRequest = new TsgcOpenAIClass_Request_Edit();
try
{
    oRequest->Model = "text-davinci-edit-001";
    oRequest->Input = "What day of the wek is it?";
    oRequest->Instruction = "Fix the spelling mistakes";
    oRequest->Temperature = 1;
    oRequest->N = 2;
    TsgcOpenAIClass_Response_Edit *oResponse = OpenAI->CreateEdit(oRequest);

    if (Length(oResponse->Choices) > 0)
    {
        ShowMessage(oResponse->Choices[0]->Text);
    }
}
__finally
{
    delete oRequest;
    delete oResponse;
}
```

OpenAI | 音频

创建转录

将音频转录为输入语言。

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

ShowMessage(OpenAI->_CreateTranscriptionFromFile("whisper-1", "c:\\media\\audio.mp3"));
```

创建翻译

将音频翻译为英语。

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

ShowMessage(OpenAI->_CreateTranslationFromFile("whisper-1", "c:\\media\\audio.mp3"));
```

OpenAI | 内容审核

给定输入文本后，输出模型是否将其分类为违反 OpenAI 内容政策。

简单示例

对以下文本进行内容审核

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

ShowMessage(OpenAI->_CreateModeration("I want to kill them."));
```

高级示例

使用选定模型对以下文本进行审核。

```
TsgcHTTP_OpenAI_JSON *OpenAI = new TsgcHTTP_OpenAI_JSON(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

TsgcOpenAIClass_Request_Moderation *oRequest = new TsgcOpenAIClass_Request_Moderation();
try
{
    oRequest->Model = "text-moderation-latest";
    oRequest->Input = "I want to kill them.";
    TsgcOpenAIClass_Response_Moderation *oResponse = OpenAI->CreateModeration(oRequest);

    if (Length(oResponse->results) > 0)
    {
        ShowMessage(oResponse->results[0]->flagged);
    }
}
__finally
{
    delete oRequest;
    delete oResponse;
}
```

OpenAI | 实时

您可以将 Realtime API 用于纯转录场景，无论是从麦克风还是文件输入。例如，您可以用它实时生成字幕或转录文本。在纯转录模式下，模型不会生成响应。

要使用实时 API 进行转录，您需要创建转录会话，通过 WebSocket 连接。使用 [TsgcWSAPI_OpenAI](#) 和 [TsgcWebSocketClient](#) 组件启动新的转录会话。

以下是使用 OpenAI API 进行实时转录的示例。

```
TsgcWebSocketClient *WSClient = new TsgcWebSocketClient(NULL);
TsgcAudioRecorderWave *oAudio = new TsgcAudioRecorderWave(NULL);
TsgcWSAPI_OpenAI *OpenAI = new TsgcWSAPI_OpenAI(NULL);
OpenAI->Client = WSClient;
OpenAI->AudioRecorder = oAudio;
OpenAI->OpenAIOptions->APIKey = "your-api-key-here";
OpenAI->OpenAIOptions->Method = sgcoaimTranscription;
OpenAI->OpenAIOptions->Provider = sgcoaipOpenAI;
OpenAI->InputAudio->Language = "en";
OpenAI->InputAudio->Model = "whisper-1";
void __fastcall TForm1::OnOpenAIAudioTranscriptionCompleted(TObject *Sender, TsgcWSOpenAIConversation_Item_Comple
    Log("#transcription_completed: " + aItem->Transcript);
}
```

手动发送音频

该组件允许您手动发送音频文件。您可以使用 AppendInputAudioBuffer 方法并将音频作为 TStream 传入。音频格式必须为 24 kHz 单声道 PCM（仅支持 24000 的采样率）。

OpenAI | 响应

Responses API 是与 OpenAI 模型交互的核心 API。它支持文本和图像输入、工具使用、流式传输以及结构化输出。

简单示例

使用 Responses API 创建响应。

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";
ShowMessage(OpenAI->_CreateResponse("gpt-4o", "What is the capital of France?"));
```

方法

- **CreateResponse**：根据给定模型和输入创建模型响应。
 - **Model**：要使用的模型 ID（例如 gpt-4o、gpt-4o-mini）。
 - **Input**：模型的文本或结构化输入。
- **RetrieveResponse**：通过 ID 检索之前创建的响应。
 - **ResponseId**：要检索的响应 ID。
- **DeleteResponse**：通过 ID 删除响应。
 - **ResponseId**：要删除的响应 ID。
- **CancelResponse**：取消正在进行的响应。
 - **ResponseId**：要取消的响应 ID。
- **ListInputItems**：返回给定响应的输入项列表。
 - **ResponseId**：响应的 ID。

OpenAI | 语音

使用文本转语音（TTS）API 从文本生成语音音频。支持多种音色和输出格式。

简单示例

从文本生成语音并保存至文件流。

```
OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

TFileStream *oStream = new TFileStream("output.mp3", fmCreate);
try
{
    OpenAI->_CreateSpeech("tts-1", "Hello, how are you?", "alloy", oStream);
}
__finally
{
    delete oStream;
}
```

方法

- **CreateSpeech**：从输入文本生成音频。
 - **Model**：要使用的模型 ID（tts-1 或 tts-1-hd）。
 - **Input**：要生成音频的文本，最多 4096 个字符。
 - **Voice**：要使用的声音（alloy、echo、fable、onyx、nova、shimmer）。
 - **ResponseStream**：写入音频数据的流。
 - **ResponseFormat**：（可选）音频格式：mp3（默认）、opus、aac 或 flac。
 - **Speed**：（可选）生成音频的速度（0.25 至 4.0，默认 1.0）。

OpenAI | 微调

管理微调作业，使模型适应您的特定训练数据。微调让您通过提供更高质量的结果、更多训练样本的能力以及降低成本，从模型中获得更多价值。

简单示例

创建微调作业并列出现有作业。

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

// Create a fine-tuning job
ShowMessage(OpenAI->_CreateFineTuningJob("gpt-4o-mini-2024-07-18", "file-abc123"));

// List fine-tuning jobs
ShowMessage(OpenAI->_ListFineTuningJobs());
```

方法

- **CreateFineTuningJob**：创建一个微调任务，开始从给定数据集创建新模型的过程。
 - **Model**：要微调的模型名称。
 - **TrainingFile**：包含训练数据的已上传文件的 ID。
- **ListFineTuningJobs**：列出您的组织的微调任务。
- **RetrieveFineTuningJob**：获取微调任务的相关信息。
 - **JobId**：微调任务的 ID。
- **CancelFineTuningJob**：立即取消一个微调任务。
 - **JobId**：要取消的微调任务的 ID。
- **ListFineTuningJobEvents**：获取微调任务的状态更新。
 - **JobId**：微调任务的 ID。
- **ListFineTuningJobCheckpoints**：列出微调任务的检查点。
 - **JobId**：微调任务的 ID。

OpenAI | 批处理

创建大批量 API 请求以异步运行。Batch API 在 24 小时内返回完成结果，享有 50% 折扣。

简单示例

创建一个批处理作业并获取其状态。

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

// Create a batch
ShowMessage(OpenAI->_CreateBatch("file-abc123", "/v1/chat/completions"));

// List batches
ShowMessage(OpenAI->_ListBatches());
```

方法

- **CreateBatch**：从已上传的请求文件创建并执行批处理。
 - **InputFileId**：包含批处理请求的已上传文件的 ID。
 - **Endpoint**：用于批次中所有请求的端点（例如 /v1/chat/completions）。
 - **CompletionWindow**：（可选）批次应在其中处理完成的时间范围。默认值为 24h。
- **RetrieveBatch**：通过 ID 检索批次。
 - **BatchId**：要检索的批次 ID。
- **ListBatches**：列出您的组织的批处理任务。
- **CancelBatch**：取消正在进行的批处理。
 - **BatchId**：要取消的批处理的 ID。

OpenAI | 上传

分段上传大文件。Uploads API 允许您通过将文件拆分为多个部分并分别上传每个部分来上传大于 512 MB 的文件。

简单示例

创建上传任务，添加分块，然后完成上传。

```
TsgcHTTP_API_OpenAI *OpenAI = new TsgcHTTP_API_OpenAI(NULL);
OpenAI->OpenAIOptions->ApiKey = "API_KEY";

// Create an upload
AnsiString vUploadResponse = OpenAI->CreateUpload("training_data.jsonl",
    "fine-tune", 2147483648, "application/jsonl");

// Add a part
AnsiString vPartResponse = OpenAI->AddUploadPart("upload_abc123", "part1.jsonl");

// Complete the upload
ShowMessage(OpenAI->CompleteUpload("upload_abc123", "[\"part_abc123\"]"));
```

方法

- **CreateUpload**：创建中间 Upload 对象以开始分段上传。
 - **Filename**：要上传的文件名。
 - **Purpose**：已上传文件的预期用途（例如 fine-tune）。
 - **Bytes**：文件的字节数。
 - **MimeType**：文件的 MIME 类型（例如 application/jsonl）。
- **AddUploadPart**：向 Upload 对象添加一个部分。
 - **UploadId**：Upload 的 ID。
 - **Filename**：此部分的字节块。
- **CompleteUpload**：完成上传。上传的文件仅在完成后方可使用。
 - **UploadId**：上传的 ID。
 - **PartIds**：以 JSON 数组字符串表示的有序 Part ID 列表。
- **CancelUpload**：取消上传。取消后不能再添加任何部分。
 - **UploadId**：要取消的上传 ID。

OpenAI 应用程序

概述

使用 OpenAI API，您可以构建各种各样的应用程序，以下是一些示例：

- **聊天机器人与虚拟助手**：能够以自然、类人方式与人类对话的应用程序。可用于客户支持、处理查询以及在网站或移动应用上提供信息。
- **内容生成**：能够生成类人文本（如文章、博客文章或报告）的应用程序。例如，GPT-3 可用于自动化社交媒体内容创作、生成代码或创建 SEO 友好的内容。
- **翻译服务**：可以将文本从一种语言翻译成另一种语言的应用程序。
- **辅导与教育**：AI 可用于创建个性化学习体验、帮助完成家庭作业或用简单语言解释复杂概念。
- **游戏**：OpenAI 可用于创建沉浸式交互游戏，尤其是涉及对话角色或复杂叙事的游戏。
- **情感分析**：分析和分类文本数据中表达的情感，可用于市场研究、品牌监测和了解客户反馈。
- **个性化推荐**：AI 可根据用户的历史行为和偏好，为产品、服务或内容生成个性化推荐。
- **文本补全**：在电子邮件客户端或文字处理软件中补全用户句子或辅助写作。
- **语音识别**：将口语转录为书面文字，适用于转录服务、语音助手等场景。
- **医疗和法律建议**：虽然 AI 模型无法替代专业建议，但可以根据给定的输入提供初步指导或建议。

组件

以下是可用组件列表及其简短描述。

- **TsgcAIOpenAIAssistant**：助手具有指令，可以利用模型、工具和文件来响应用户查询。
- **TsgcAIOpenAIChatBot**：一个聊天机器人，通过 OpenAI Whisper API 监听语音并将其转换为文本。该文本发送至 ChatCompletion API，由 OpenAI 返回响应，再将响应从文本转换为语音。
- **TsgcAIOpenAITranslator**：一个翻译应用程序，允许您将任何语言的语音翻译成英语，并使用任何可用的 TextToSpeech 组件收听翻译。
- **TsgcAIOpenAIEmbeddings**：此组件用于构建具有自定义数据的 AI 应用程序，例如：包含产品数据的聊天机器人。支持以下数据库：
 - **TsgcAIDatabaseVectorPinecone**：支持 Pinecone 向量数据库。

以下组件用于从麦克风录制音频、播放音频文件以及将文本转换为语音。

- **TsgcAudioRecorderMCI**：（仅限 Windows）此组件允许您访问麦克风并将语音转换为波形文件。
- **TsgcAudioPlayerMCI**：（仅适用于 Windows）此组件允许您播放 mp3 文件。
- **TsgcTextToSpeechSystem**：（仅限 Windows）无需使用外部 mp3 文件即可将文本转换为语音。
- **TsgcTextToSpeechGoogle**：使用任意 Google Cloud 语音将文本转换为语音。
- **TsgcTextToSpeechAmazon**：使用任意 Amazon AWS 语音将文本转换为语音。

TsgcAIOpenAIAssistant

Assistants API 允许您在自己的应用程序中构建 AI 助手。

简介

Assistants API 允许您在自己的应用程序中构建 AI 助手。助手具有指令，可以利用模型、工具和文件来响应用户查询。Assistants API 目前支持三种工具：代码解释器、文件搜索和函数调用。

概述

Assistants API 的典型集成流程如下：

- 通过定义自定义指令并选择模型来创建助手。如有需要，可添加文件并启用代码解释器、文件搜索和函数调用等工具。
- 当用户开始对话时创建一个线程。
- 随着用户提问，将消息添加到 Thread。
- 通过调用模型和工具，在线程上运行助手以生成响应。

步骤 1：创建助手

助手代表一个可以配置为响应用户消息的实体，可使用模型、指令和工具等多个参数进行配置。

```
// Create a new assistant
TsgcAIOpenAIAssistant *oAssistant = new TsgcAIOpenAIAssistant(NULL);
// Set your API key
oAssistant->OpenAIOptions->ApiKey = txtAPIKey->Text;
// Assistant options
oAssistant->AssistantOptions->Name = "Math Tutor";
oAssistant->AssistantOptions->Instructions->Text = "You are a personal math tutor. Write and run code to answer questions";
oAssistant->AssistantOptions->Model = "gpt-4o";
// Create the assistant
oAssistant->CreateAssistant();
```

步骤 2：创建线程

Thread 代表用户与一个或多个 Assistant 之间的对话。您可以在用户（或您的 AI 应用程序）开始与 Assistant 对话时创建一个 Thread。

```
oThread = oAssistant->CreateThread();
```

步骤 3：向线程添加消息并运行

您的用户或应用程序创建的消息内容作为 Message 对象添加到 Thread 中。消息可包含文本和文件。您可以向 Thread 添加的消息数量没有限制——我们会智能截断不适合模型上下文窗口的任何上下文。

将所有用户消息添加到线程后，您可以使用任何助手运行该线程。创建运行将使用与助手关联的模型和工具来生成响应。这些响应将作为助手消息添加到线程中。

```
void SendMessage(TsgcAIOpenAIAssistant *oAssistant, TsgcAIClass_Thread *oThread, const UnicodeString aMessage)
{
    int i;
    TsgcOpenAIClass_Message *oMessage;
    TsgcOpenAIClass_Response_List_Messages *oMessages;
    TsgcOpenAIClass_Run *oRun;
    DoLog(L"[user]: " + aMessage);
    oMessage = oAssistant->CreateMessageText(oThread->Id, aMessage);
    if (oMessage != NULL)
    {
        oRun = oAssistant->CreateRunAndWait(oThread->Id);
        if (oRun != NULL)
        {
            oMessages = oAssistant->GetMessages(oThread->Id, oRun->Id);
            if (oMessages != NULL && oMessages->Messages.Length > 0)
            {
                for (i = 0; i < oMessages->Messages.Length; i++)
                {
                    DoLog(L"[assistant]: " + DoFormatResponse(oMessages->Messages[i]->ContentText + L"\r\n"));
                }
            }
        }
    }
}
```

参考

- 属性 — 组件上提供 5 个属性。
- 方法——9 个公开方法。
- 事件 — 8 个可处理的事件。

TsgcAIOpenAIAssistant | 文件搜索

文件搜索通过集成来自外部来源的知识来增强 Assistant 的能力。其工作方式如下：

- **文档处理**：OpenAI 自动处理并将您的文档分割（“分块”）为可管理的信息片段。
- **嵌入创建**：这些数据块被转换为嵌入（即内容的数学表示），以捕获其含义和上下文。
- **存储与搜索**：系统存储这些嵌入，同时支持向量搜索（查找语义相似内容）和关键字搜索（定位精确匹配项）。

这种组合确保助手能够检索并利用相关内容，为用户查询提供准确且有据可查的答案。

步骤 1：创建一个启用文件搜索的新助手

在助手的 tools 参数中启用 file_search，创建一个新助手。启用 file_search 工具后，模型将根据用户消息决定何时检索内容。

```
TsgcAIOpenAIAssistant* Assistant = new TsgcAIOpenAIAssistant(nullptr);
Assistant->OpenAIOptions->ApiKey = "sk-askdjfalskdjfl23kjkjasdefasdj";
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;
```

步骤 2：上传文件并将其添加到向量存储

要访问您的文件，file_search 工具使用向量存储对象。上传您的文件并创建向量存储以容纳它们。

```
void UploadFile()
{
    TOpenDialog* oDialog = new TOpenDialog(nullptr);
    try
    {
        if (oDialog->Execute())
        {
            Screen->Cursor = crHourGlass; // Change cursor to hourglass
            try
            {
                Assistant->UploadVectorStoreFile("sgcVectorStore", oDialog->FileName.c_str());
            }
            finally
            {
                Screen->Cursor = crDefault; // Reset cursor to default
            }
        }
    }
    finally
    {
        delete oDialog; // Clean up dialog
    }
}
```

步骤 3：创建运行并检查输出

现在，创建一次运行（Run），观察模型如何使用 File Search 工具来响应用户的问题。

```
void SendMessage()
{
    int i;
    TsgcOpenAIClass_Message* oMessage = nullptr;
    TsgcOpenAIClass_Response_List_Messages* oMessages = nullptr;
    TsgcOpenAIClass_Run* oRun = nullptr;
    DoLog("[user]: " + memoMessage->Lines->Text);
    Screen->Cursor = crHourGlass; // Change cursor to hourglass
    try
    {
        oMessage = Assistant->CreateMessageText("thread_id", "Create a WebSocket Client that connects to eSeGeCe");
        if (oMessage != nullptr)
        {
            oRun = Assistant->CreateRunAndWait("thread_id");
            if (oRun != nullptr)
            {
                oMessages = Assistant->GetMessages("thread_id", oRun->Id);
                if (oMessages != nullptr && oMessages->Messages.Length > 0)
                {
                    memoMessage->Lines->Clear();
                    for (i = 0; i < oMessages->Messages.Length; i++)
                    {
                        DoLog("[assistant]: " + DoFormatResponse(oMessages->Messages[i].ContentText + "\r\n"));
                    }
                }
            }
        }
    }
    __finally
    {
        Screen->Cursor = crDefault; // Reset cursor to default
    }
}
```

TsgcAIOpenAIAssistant | 流式传输

您无需等待助手的完整响应，可以使用 Server-Sent Events 以流式方式接收响应。在使用 **CreateRun** 函数时只需传入参数 **Stream = True**，响应即可使用流式传输。

以下事件用于处理流式响应：

- **OnStreamRun**：当 run 对象有更新时调用该事件。
- **OnStreamMessage**：当消息对象有更新时调用此事件：已创建、进行中、已完成.....
- **OnStreamMessageDelta**：当消息的部分内容正在流式传输时发生。
- **OnStreamDone**：当流结束时发生。
- **OnStreamError**：发生错误时触发。可能由服务器内部错误或超时引起。

步骤 1：创建助手

助手代表一个可以配置为响应用户消息的实体，可使用模型、指令和工具等多个参数进行配置。

```
// Create a new assistant
TsgcAIOpenAIAssistant *oAssistant = new TsgcAIOpenAIAssistant(NULL);
// Set your API key
oAssistant->OpenAIOptions->ApiKey = txtAPIKey->Text;
// Assistant options
oAssistant->AssistantOptions->Name = "Math Tutor";
oAssistant->AssistantOptions->Instructions->Text = "You are a personal math tutor. Write and run code to answer n";
oAssistant->AssistantOptions->Model = "gpt-4o";
// Create the assistant
oAssistant->CreateAssistant();
```

步骤 2：创建线程

Thread 代表用户与一个或多个 Assistant 之间的对话。您可以在用户（或您的 AI 应用程序）开始与 Assistant 对话时创建一个 Thread。

```
oThread = oAssistant->CreateThread();
```

步骤 3：向 Thread 添加消息并使用流式传输运行

您的用户或应用程序创建的消息内容作为 Message 对象添加到 Thread 中。消息可包含文本和文件。您可以向 Thread 添加的消息数量没有限制——我们会智能截断不适合模型上下文窗口的任何上下文。

将所有用户消息添加到线程后，您可以使用任意助手运行该线程。创建运行会使用与助手关联的模型和工具生成响应。响应通过服务器推送流发送。

```
void SendMessage(TsgcAIOpenAIAssistant *oAssistant, TsgcAIClass_Thread *oThread, const UnicodeString aMessage)
{
    int i;
    TsgcOpenAIClass_Message *oMessage;
    TsgcOpenAIClass_Response_List_Messages *oMessages;
    TsgcOpenAIClass_Run *oRun;
    DoLog(L"[user]: " + aMessage);
    oMessage = oAssistant->CreateMessageText(oThread->Id, aMessage);
    if (oMessage != NULL)
```

```
{
    oRun = oAssistant->CreateRun(oThread->Id, true);
}
```

第 4 步：处理响应

使用 OnStreamMessageDelta 事件读取服务器发送的流式消息。

```
void __fastcall OnStreamMessageDelta(TObject *Sender, const TsgcOpenAIClass_MessageDelta &aMessageDelta, const St
{
    for (int i = aMessageDelta.Content.Low(); i <= aMessageDelta.Content.High(); i++)
    {
        String vType = aMessageDelta.Content[i]->_Type;
        if (vType == "text")
        {
            String vResponse = static_cast<TsgcOpenAIClass_MessageDeltaContent_Text*>(aMessageDelta.Content[i])->
        }
    }
}
```

TsgcAIOpenAIAssistant 函数调用

与聊天完成 API 类似，Assistants API 支持函数调用。函数调用允许您向 Assistants API 描述函数，并让其智能地返回需要调用的函数及其参数。

在此示例中，我们将创建一个天气助手，并定义两个函数 `get_current_temperature` 和 `get_rain_probability`，作为助手可以调用的工具。在使用并行函数调用的示例中，我们将询问助手今天旧金山的天气如何以及降雨概率。我们还展示了如何以流式方式输出助手的响应。

步骤 1 定义函数

创建助手时，您将首先在助手的 `tools` 参数下定义函数。

```
TsgcAIOpenAIAssistant *Assistant = new TsgcAIOpenAIAssistant(nullptr);
Assistant->OpenAIOptions->ApiKey = "sk-askdjfalskdjfl23kjkjasdefasdj";
Assistant->AssistantOptions->Name = "Delphi Weather Bot";
Assistant->AssistantOptions->Instructions->Text = "You are a weather bot. Use the provided functions to answer qu";
Assistant->AssistantOptions->Model = "gpt-4o";
Assistant->AssistantOptions->Tools->Functions->Enabled = false;
Assistant->AssistantOptions->Tools->Functions->Functions->Text =
    "[{\"type\":\"function\",\"function\":{\"name\":\"get_current_temperature\",\"description\":\"Get the current
Assistant->AssistantOptions->Tools->FileSearch->Enabled = false;
Assistant->AssistantOptions->Tools->CodeInterpreter->Enabled = false;
```

步骤 2：创建线程并添加消息

当用户开始对话时创建 Thread，并随着用户提问在 Thread 中添加 Message。

```
void SendMessage()
{
    int i;
    TsgcOpenAIClass_Message* oMessage = nullptr;
    TsgcOpenAIClass_Response_List_Messages* oMessages = nullptr;
    TsgcOpenAIClass_Run* oRun = nullptr;
    DoLog("[user]: " + memoMessage->Lines->Text);
    Screen->Cursor = crHourGlass; // Change cursor to hourglass
    try
    {
        oMessage = Assistant->CreateMessageText("thread_id", "What is the weather in San Francisco today and the
        if (oMessage != nullptr)
        {
            oRun = Assistant->CreateRunAndWait("thread_id");
            if (oRun != nullptr)
            {
                oMessages = Assistant->GetMessages("thread_id", oRun->Id);
                if (oMessages != nullptr && oMessages->Messages.Length > 0)
                {
                    memoMessage->Lines->Clear();
                    for (i = 0; i < oMessages->Messages.Length; i++)
                    {
                        DoLog("[assistant]: " + DoFormatResponse(oMessages->Messages[i].ContentText + "\r\n"));
                    }
                }
            }
        }
    }
    finally
    {
        Screen->Cursor = crDefault; // Reset cursor to default
    }
}
```

步骤 3：处理 OnFunctionCall 事件

当组件检测到需要函数参数值时，会调用 **OnFunctionCall** 事件。使用 **Request._Function** 参数了解请求详情，并使用 **Response.Output** 发送响应。

```
void __fastcall TFRMOpenAIAssistant::AssistantFunctionCall(TObject *Sender,
    const TsgcOpenAIClass_ToolCall &aRequest,
    const TsgcHTTPOpenAI_ToolCall_Response &aResponse)
{
    if (aRequest.Function.Name == "get_current_temperature")
        aResponse.Output = 30;
    else if (aRequest.Function.Name == "get_rain_probability")
        aResponse.Output = 10;
}
```

OpenAI Audio

要将 OpenAI API 与语音命令配合使用，需要以下步骤：

1. 需要捕获麦克风音频，因此需要一个语音转文字系统将其转换为文本后再发送至 OpenAI。
 1. 使用组件 [TsgcAudioRecorderMCI](#) 捕获麦克风音频。
 2. 捕获音频后，将其发送至 OpenAI Whisper API，将音频文件转换为文本。
2. 获取语音转文字结果后，使用 ChatCompletion API 将文本发送给 OpenAI。
3. OpenAI 的响应随后须通过以下组件之一转换为语音：
 1. [TsgcTextToSpeechSystem](#)：（目前仅支持 Windows）使用操作系统的 Windows 文字转语音功能。
 2. [TsgcTextToSpeechGoogle](#)：将 OpenAI 的响应发送至 Google Cloud 服务器，返回一个 MP3 文件，由 [TsgcAudioPlayerMCI](#) 播放。
 3. [TsgcTextToSpeechAmazon](#)：将 OpenAI 的响应发送至 Amazon AWS 服务器，返回一个 MP3 文件，由 [TsgcAudioPlayerMCI](#) 播放。

组件

以下组件用于从麦克风录制音频、播放音频文件以及将文本转换为语音。

- [TsgcAudioRecorderMCI](#)：（仅限 Windows）此组件允许您访问麦克风并将语音转换为 Wave 文件。
- [TsgcAudioPlayerMCI](#)：（仅限 Windows）此组件允许您播放 mp3 文件。
- [TsgcTextToSpeechSystem](#)：（仅限 Windows）无需使用外部 mp3 文件即可将文本转换为语音。
- [TsgcTextToSpeechGoogle](#)：使用任意 Google Cloud 语音将文本转换为语音。
- [TsgcTextToSpeechAmazon](#)：使用 Amazon AWS 提供的任意语音将文本转换为语音。

TsgcAudioRecorderMCI

此组件用于捕获麦克风音频并将其存储为 wave 文件。

简介

此组件用于捕获麦克风音频并将其存储为 wave 文件。目前仅支持 Windows。

参考

- 属性 — 组件上提供 3 个属性。
- 方法 — 2 个公共方法。
- 事件——0 个可处理的事件。

TsgcAudioPlayerMCI

此组件用于播放从文本转语音提供商收到的 mp3 文件。

简介

此组件用于播放从文本转语音提供商接收到的 mp3 文件。目前仅支持 Windows。

参考

- 属性 — 组件上有 1 个属性可用。
- 方法 — 1 个公开方法。
- Events — 0 个可处理的事件。

TsgcTextToSpeechSystem

这是操作系统提供的默认文字转语音功能，目前仅支持 Windows。

简介

这是操作系统提供的默认文字转语音功能，目前仅支持 Windows。

参考

- 属性 — 组件上提供 1 个属性。
- 方法 — 1 个公共方法。
- 事件 — 0 个可处理的事件。

TsgcTextToSpeechGoogle

Text-To-Speech 是 Google Cloud 提供的 API，允许您将文本转换为 mp3 文件。

简介

Text-To-Speech 是 Google Cloud 提供的 API，允许您将文本转换为 mp3 文件。需要 Google Cloud 账户并设置 Text-To-Speech 账户。

配置好文字转语音账户后，须下载一个 JSON 设置文件，并将其设置到 `GoogleOptions.Settings` 属性中。

参考

- 属性 — 该组件提供 3 个属性。
- 方法 — 1 个公共方法。
- 事件 — 0 个可处理的事件。

TsgcTextToSpeechAmazon

Text-To-Speech 是 Amazon AWS 提供的 API，可将文本转换为 mp3 文件。

简介

Text-To-Speech 是 Amazon AWS 提供的一种 API，可将文本转换为 mp3 文件。它需要 Amazon AWS 账户并设置 Polly API。

参考

- 属性 — 组件上共有 3 个可用属性。
- 方法 — 1 个公开方法。
- 事件 — 可处理 0 个事件。

TsgcAIChat

TsgcAIChat 是一个可与任何 AI 提供商配合使用的单一组件。

简介

TsgcAIChat 是一个单一组件，可与任何 AI 提供程序配合使用。通过更改一个属性即可切换提供程序。该组件内置会话历史管理功能，无需手动跟踪消息即可轻松构建聊天应用程序。

支持的提供商

- OpenAI
- Anthropic
- Gemini
- DeepSeek
- Ollama
- Grok
- Mistral

参考

- Properties — 组件上有 5 个可用属性。
- 方法 — 4 个公共方法。
- 事件 — 3 个可处理的事件。

TsgcAIOpenAIChatBot

必须捕获麦克风音频，因此需要一个语音转文字系统来获取将发送给 OpenAI 的文本。

简介

要构建具有语音命令的聊天机器人，需要以下步骤：

1. 需要捕获麦克风音频，因此需要一个语音转文字系统来获取将发送给 OpenAI 的文本。
 1. 使用组件 [TsgcAudioRecorderMCI](#) 捕获麦克风音频。
 2. 捕获音频后，将其发送至 OpenAI Whisper API，将音频文件转换为文本。
2. 获取语音转文字结果后，使用 ChatCompletion API 将文本发送给 OpenAI。
3. 来自 OpenAI 的响应随后必须使用以下组件之一转换为语音：
 1. [TsgcTextToSpeechSystem](#)：（目前仅限 Windows）使用操作系统的 Windows 文本转语音功能。
 2. [TsgcTextToSpeechGoogle](#)：将 OpenAI 的响应发送到 Google Cloud 服务器，返回一个由 [TsgcAudioPlayerMCI](#) 播放的 MP3 文件。
 3. [TsgcTextToSpeechAmazon](#)：将 OpenAI 的响应发送到 Amazon AWS 服务器，返回一个由 [TsgcAudioPlayerMCI](#) 播放的 MP3 文件。

参考

- 属性 — 该组件提供 6 个属性。
- 方法 — 4 个公共方法。
- 事件 — 4 个可处理的事件。

TsgcAIOpenAITranslator

必须捕获麦克风音频，因此需要一个语音转文字系统来获取将发送给 OpenAI 的文本。

简介

要构建带有语音命令的翻译器，需要执行以下步骤：

1. 需要捕获麦克风音频，因此需要一个语音转文字系统来获取将发送给 OpenAI 的文本。
 1. 使用组件 [TsgcAudioRecorderMCI](#) 捕获麦克风音频。
 2. 捕获音频后，将其发送至 OpenAI Whisper API，将音频文件转换为文本。
2. 获取语音转文字结果后，使用 ChatCompletion API 将文本发送给 OpenAI。
3. 来自 OpenAI 的响应随后必须使用以下组件之一转换为语音：
 1. [TsgcTextToSpeechSystem](#)：（目前仅限 Windows）使用操作系统的 Windows 文本转语音功能。
 2. [TsgcTextToSpeechGoogle](#)：将 OpenAI 的响应发送到 Google Cloud 服务器，返回一个由 [TsgcAudioPlayerMCI](#) 播放的 MP3 文件。
 3. [TsgcTextToSpeechAmazon](#)：将 OpenAI 的响应发送到 Amazon AWS 服务器，返回一个由 [TsgcAudioPlayerMCI](#) 播放的 MP3 文件。

参考

- 属性——组件上提供 5 个属性。
- 方法 — 2 个公共方法。
- 事件 — 3 个可处理的事件。

TsgcAIOpenAIEmbeddings

嵌入是一种以数字形式表示单词、短语或其他类型数据（如图像或音频）的方式。

简介

嵌入是一种将单词、短语甚至图像或音频等其他类型数据以数字形式表示的方法。就像将单词或数据转化为数字，使计算机能够更好地理解和处理它们。

设想您有一组词语，例如"dog"（狗）、"cat"（猫）和"bird"（鸟）。这些词是有含义的，对吗？嵌入（Embeddings）为每个词分配一组唯一的数字（向量），用于捕捉其含义及与其他词的关系。

例如，"dog"这个词可能被表示为 [0.5, 0.2, -0.7]，"cat"表示为 [0.8, -0.3, 0.1]，"bird"表示为 [0.3, 0.9, 0.4]。向量中的数字携带每个词的特征信息，例如它们是否与动物相关或彼此的相似程度。

令人惊叹的是，嵌入可以从大量数据中学习，因此它们可以自动找出单词之间的相似性和差异性。这些数值表示帮助 AI 算法理解语言并理解复杂模式，这在语言翻译、情感分析和推荐系统等各种应用中至关重要。它们还使 AI 模型更容易、更快速地处理信息并提供更准确的结果！

参考

- 属性 — 组件上提供 4 个属性。
- 方法——2 个公共方法。
- 事件 — 5 个可处理的事件。

TsgcAIDatabaseVectorFile

该组件将数据库向量和提示存储在两个文本文件中。

简介

该组件将数据库向量和提示存储在两个文本文件中。此组件仅应用于测试目的，不适合生产环境，因为当向量数量较多时性能未经优化。

配置

- **VectorFileOptions** :
 - **InputFilename** : 用于存储输入数据的文件名。
 - **VectorFilename** : 用于存储向量的文件名。

参考

- 属性 — 组件上提供 2 个属性。
- 方法 — 4 个公共方法。
- 事件 — 0 个可处理的事件。

TsgcAIDatabaseVectorPinecone

TsgcAIDatabaseVectorPinecone 组件。

简介

该组件基于 REST [Pinecone API 客户端](#)，允许您创建/更新/删除索引和向量。

配置

- **PineconeOptions** :
 - **ApiKey**：在此配置 Pinecone 提供的 API Key，可从您的 Pinecone 账户获取。
 - **Environment**：默认为免费账户的"us-west4-gcp-free"。
 - **LogOptions**：在此配置是否将 HTTP 请求存储到文本文件。
- **PineconeIndexOptions** :
 - **IndexName**：用于存储或查询数据的索引名称。
 - **ProjectId**：项目的 ID。

参考

- 属性 — 组件上提供 3 个属性。
- 方法 — 4 个公共方法。
- 事件——共 0 个可处理事件。

嵌入 | 创建向量

要使用嵌入，首先需要将数据转换为向量。

示例

如果您有 PDF 文件，请先将其转换为文本，然后使用 **CreateEmbeddingsFromFile** 方法获取向量数据。由于 OpenAI 嵌入的大小限制，如果文件过大，数据将自动拆分为多个块，因此一个文件可以生成一个或多个向量。

以下是代码示例。

```
void ConvertFileToVector()
{
    TOpenDialog* oDialog = new TOpenDialog(NULL);
    try
    {
        oDialog->Filter = "TXT Files|*.txt";
        if (oDialog->Execute())
        {
            TsgcAIOpenAIEmbeddings* oEmbeddings = new TsgcAIOpenAIEmbeddings(NULL);
            try
            {
                TsgcAIDatabaseVectorFile* oFile = new TsgcAIDatabaseVectorFile(NULL);
                try
                {
                    oEmbeddings->Database = oFile;
                    oEmbeddings->OpenAIOptions->ApiKey = "<your api key>";
                    oEmbeddings->CreateEmbeddingsFromFile(oDialog->FileName);
                }
                __finally
                {
                    delete oFile;
                }
            }
            __finally
            {
                delete oEmbeddings;
            }
        }
    }
    __finally
    {
        delete oDialog;
    }
}
```

嵌入 | ChatBot

将所有数据转换为向量后，我们就可以开始构建自己的模型了。其背后的思路非常简单：每次提问时，首先将问题转换为向量，然后在数据库中搜索与问题最相似的向量，最后使用最相似的数据作为上下文。

```
void AskToChatGPT(const std::string& aQuestion)
{
    TsgcAIOpenAIChatBot* oChatBot = new TsgcAIOpenAIChatBot(NULL);
    try
    {
        oChatBot->OpenAIOptions->ApiKey = "<your api key>";
        TsgcAIOpenAIEmbeddings* oEmbeddings = new TsgcAIOpenAIEmbeddings(NULL);
        try
        {
            oChatBot->Embeddings = oEmbeddings;
            TsgcAIDatabaseVectorFile* oFile = new TsgcAIDatabaseVectorFile(NULL);
            try
            {
                oEmbeddings->Database = oFile;
                std::string vContext = oChatBot->GetEmbedding(aQuestion);
                std::string message = "Answer the question based on the context below.\n\nContext:\n" +
                    vContext + "\nQuestion:" + aQuestion + "\nAnswer:";
                oChatBot->ChatAsUser(message.c_str());
            }
            __finally
            {
                delete oFile;
            }
        }
        __finally
        {
            delete oEmbeddings;
        }
    }
    __finally
    {
        delete oChatBot;
    }
}
```

Pinecone

[Pinecone.io](https://pinecone.io)

Pinecone 是一个向量数据库，允许您以简单而强大的方式上传、查询和删除向量数据。

Pinecone 拥有一个公共 API，允许第三方将 Pinecone 集成到自己的应用程序中。组件 `TsgcHTTP_API_Pinecone` 是 Pinecone API 的封装器。

配置

开始之前，您必须在 Pinecone 网站上注册并申请 API 密钥。此 API 密钥用于发送 API 请求，必须在 `TsgcHTTP_API_Pinecone` 组件的 **`PineconeOptions.ApiKey`** 属性中进行设置。

索引操作

支持以下方法：

方法	参数	描述
<code>IndexesList</code>		此操作返回您的 Pinecone 索引列表。
<code>IndexCreate</code>	<code>TsgcHTTTPineconeIndexCreate</code>	此操作创建一个 Pinecone 索引。您可以使用它来指定相似度度量、索引中存储的向量维度、使用的副本数量等。
<code>IndexDescribe</code>	索引名称	获取某个指数的描述信息。
<code>IndexDelete</code>	索引名称	此操作删除现有索引。
<code>IndexConfigure</code>	索引名称、副本数、PodType	此操作指定索引的 Pod 类型和副本数量。

集合操作

支持以下方法：

方法	参数	描述
<code>CollectionsList</code>		此操作返回您的 Pinecone 集合列表。
<code>CollectionCreate</code>	Collection Name, Source	此操作将创建一个 Pinecone 集合。
<code>CollectionDescribe</code>	集合名称	获取集合的描述。
<code>CollectionDelete</code>	集合名称	此操作删除一个已有的集合。

向量操作

支持以下方法：

方法	参数	描述

VectorsDescribeIndexStats	Index Name, Project Id, Filter	DescribeIndexStats 操作返回有关索引内容的统计信息，包括每个命名空间的向量数量和维度数量。
VectorsQuery	Index Name, Project Id, Params	查询操作使用查询向量搜索命名空间。它检索命名空间中最相似条目的 ID 及其相似度得分。
VectorsDelete	Index Name, Project Id, Params	删除操作按 ID 从单个命名空间中删除向量。您可以按 ID 从单个命名空间中删除项目。
VectorsFetch	索引名称, 项目 ID, Ids	Fetch 操作通过 ID 从单个命名空间中查找并返回向量。返回的向量包含向量数据和/或元数据。
VectorsUpdate	Index Name, Project Id, Params	更新操作更新命名空间中的一个向量。如果包含值，它将覆盖之前的值。如果包含 set_metadata，其中指定字段的值将被添加或覆盖之前的值。
VectorsUpsert	Index Name, Project Id, Params	Upsert 操作将向量写入命名空间。如果对已有向量 ID upsert 了新值，将覆盖之前的值。

UPSERT 示例

以下示例展示如何对 Id = "id1" 的单个向量执行 UPSERT 操作。

```
void UpsertPinecone(const String aIndexName, const String aProjectId, const std::vector<double> aVector)
{
    TsgcHTTP_API_Pinecone* oPinecone = new TsgcHTTP_API_Pinecone(NULL);
    try
    {
        oPinecone->PineconeOptions.API = "your-api-key";
        TsgcHTTTPineconeVectorUpserts* oParams = new TsgcHTTTPineconeVectorUpserts();
        try
        {
            TsgcArrayOfVectorUpsert oVectors;
            oVectors.push_back(new TsgcHTTTPineconeVectorUpsert());
            oVectors[0]->Id = "id1";
            oVectors[0]->Values = aVector;
            oParams->Vectors = oVectors;
            oPinecone->VectorsUpsert(aIndexName, aProjectId, oParams);
        }
        finally
        {
            oParams->Free();
        }
    }
    finally
    {
        oPinecone->Free();
    }
}
```

示例 QUERY

以下是查询单个向量的示例。

```
void QueryPinecone(const string aIndexName, const string aProjectId, const std::vector<double>& aVector)
{
    TsgcHTTTPineconeVectorQuery* oParams = new TsgcHTTTPineconeVectorQuery();
    try
    {
        oParams->Vector = aVector;
        Pinecone.VectorsQuery(aIndexName, aProjectId, oParams);
    }
    __finally
    {
        oParams->Free();
    }
}
```

Anthropic Claude

Anthropic 是一家 AI 安全公司，致力于构建可靠、可解释且可引导的 AI 系统。其旗舰模型系列为 Claude，在对话、内容创作、复杂推理、编程等方面表现出色。

sgcWebSockets 库提供 Delphi 组件 **TsgcHTTP_API_Anthropic**，用于与 Anthropic Claude API 进行交互。

Anthropic API

Anthropic API 提供对 Claude 模型的访问，用于构建 AI 驱动的应用程序。该 API 支持文本生成、视觉（图像理解）、工具使用（函数调用）、扩展思考、文档/PDF 处理、提示缓存、引用、网络搜索、流式传输、令牌计数和消息批处理。

功能特性

- 消息
 - [Anthropic 消息示例](#)
- 视觉
 - [Anthropic 视觉示例](#)
- 工具使用
 - [Anthropic 工具使用示例](#)
- 模型
 - [Anthropic 模型示例](#)
- 批次
 - [Anthropic 批次示例](#)
- 扩展思考
 - [Anthropic 扩展思考示例](#)
- 文档
 - [Anthropic 文档示例](#)
- 提示缓存
 - [Anthropic 提示缓存示例](#)
- 引用
 - [Anthropic 引用示例](#)
- Web 搜索
 - [Anthropic Web 搜索示例](#)
- 结构化输出
 - [Anthropic 结构化输出示例](#)
- 文件
 - [Anthropic Files API 示例](#)
- MCP 连接器
 - [Anthropic MCP 连接器示例](#)

配置

Anthropic API 使用 API 密钥进行身份验证。请访问 Anthropic 控制台中的 [API 密钥](#) 页面，获取您在请求中使用的 API 密钥。

请记住，您的 API 密钥是机密！请勿与他人共享或在任何客户端代码中公开它。

此 **API Key** 必须在组件的 **AnthropicOptions.ApiKey** 属性中配置。**AnthropicOptions.AnthropicVersion** 属性指定 API 版本（默认：2023-06-01）。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "YOUR_API_KEY";
```

属性

AnthropicOptions

- **ApiKey**：用于向 Anthropic API 进行身份验证的 API 密钥。
- **AnthropicVersion**：API 版本字符串（默认：2023-06-01）。

- **BetaHeaders**：可选的 Beta 功能标头，用于启用预发布 API 特性（例如 files-api-2025-04-14、mcp-client-2025-11-20）。
- **HttpOptions**：HTTP 连接设置。
 - **ReadTimeout**：读取 HTTP 响应的超时时间（毫秒）。默认值为 0（无超时）。
- **LogOptions**：日志配置。
 - **Enabled**：为 True 时，HTTP 请求和响应将被记录到文件中。
 - **FileName**：写入日志输出的文件路径。
- **RetryOptions**：失败 API 请求的自动重试配置。
 - **Enabled**：为 True 时，失败的请求将自动重试。
 - **Retries**：最大重试次数。
 - **Wait**：两次重试之间的等待时间（毫秒）。

消息

发送包含文本和/或图像内容的结构化输入消息列表，模型将生成对话中的下一条消息。

- **CreateMessage**：使用指定的模型和参数创建消息。
 - **Model**：要使用的模型（例如 claude-sonnet-4-20250514）。
 - **Message**：用户消息内容。
 - **MaxTokens**：要生成的最大 token 数（必填，默认值为 4096）。
- **CreateMessageWithSystem**：使用系统提示创建消息。
 - **System**：设置助手行为的系统提示。
- **CreateMessageStream**：创建后用流式传输（SSE）的消息。事件通过 OnHTTPAPISSE 事件处理程序传递。
- **CreateMessageJSON**：创建一条返回符合所提供 JSON Schema 的结构化 JSON 消息。
 - **Schema**：定义输出格式的 JSON Schema 字符串。
- **CreateMessageWithThinking**：创建启用了扩展思考功能的消息。
 - **BudgetTokens**：思考的最大令牌预算（最小值为 1024）。
- **CreateDocumentMessage**：创建带有 base64 文档（PDF 或文本）的消息。
 - **DocumentBase64**：base64 编码的文档数据。
 - **MediaType**：MIME 类型（例如 application/pdf）。

视觉

Claude 能够理解消息中以 base64 编码内容块传递的图像。

- **CreateVisionMessage**：发送带文本提示的图像。
 - **ImageBase64**：base64 编码的图像数据。
 - **MediaType**：MIME 类型（image/jpeg、image/png、image/gif、image/webp）。
 - **Prompt**：随图像附带的文本提示。

工具使用

Claude 可以使用工具（函数调用）与外部系统交互。您需要定义工具的名称、描述和输入模式，Claude 会在希望调用工具时生成 tool_use 内容块。

模型

列出并描述可用的 Claude 模型。

- **GetModels**：列出所有可用模型。
- **GetModel**：检索有关特定模型的信息。
 - **ModelId**：要检索的模型 ID。

扩展思考

扩展思考功能使 Claude 在响应前能够逐步推理，从而提升数学、编码和分析等复杂任务的质量。

- **ThinkingType**：设为 'enabled' 以激活思考模式，设为 'disabled' 以关闭。
- **ThinkingBudgetTokens**：思考的 Token 预算（最小 1024，必须小于 MaxTokens）。
- **CreateMessageWithAdaptiveThinking**：使用自适应思维创建消息，让 Claude 自动决定思维深度。

文档

Claude 可以处理以内容块形式发送的 PDF 文档和文本文件。支持 base64、文本和 URL 来源类型。

- **CreateDocumentMessage**：发送带文本提示的文档。
 - **DocumentBase64**：base64 编码的文档数据。
 - **MediaType**：MIME 类型（application/pdf、text/plain）。

提示词缓存

在 API 调用之间缓存常用上下文（系统提示、内容块、工具定义），可将缓存读取成本降低高达 90%。

- **SystemCacheControl**：在请求中设置为 True 以缓存系统提示。
- **CacheControl**：在内容块或工具定义上设置为 'ephemeral'。
- **BetaHeaders**：AnthropicOptions 中的可选测试版功能头。

引文

当启用引用功能发送文档时，Claude 会在其响应中包含来源引用。引用类型包括 char_location（文本）、page_location（PDF）、content_block_location（自定义内容）以及 web_search_result_location（网络搜索）。

网络搜索

Claude 可以使用内置的 web_search 工具搜索网络以获取实时信息。其他内置工具包括 code_execution 和 computer use。

- **CreateMessageWithWebSearch**：创建一条启用网络搜索的消息。
 - **Model**：要使用的模型。
 - **Message**：用户查询。

令牌计数

在发送消息之前统计消息中的令牌数量。

- **CountTokens**：统计消息的输入 token 数量。
 - **Model**：用于 token 计数的模型。
 - **Message**：要统计 token 的消息内容。

消息批量处理

消息批处理 API 允许您异步处理大量消息。

- **ListBatches**：列出所有消息批次。
- **GetBatch**：按 ID 检索特定批次。
- **CancelBatch**：取消仍在处理中的批次。
- **GetBatchResults**：检索已完成批次的结果。

结构化输出

强制 Claude 返回符合提供的 JSON Schema 的有效 JSON。结合 Effort 参数来控制输出质量与成本。

- **CreateMessageJSON**：创建带有 JSON 架构输出的消息。
 - **Schema**：定义输出格式的 JSON Schema 字符串。
- **OutputFormatSchema**：（Request 属性）用于结构化输出的 JSON Schema。
- **Effort**：（请求属性）控制质量与成本的平衡：'low'、'medium'、'high'、'max'。
- **Strict**：（工具属性）为 True 时，工具输入保证与 input_schema 匹配。

Files API

上传、列出、检索、下载和删除文件。上传的文件可在消息中通过带有文件源类型的文档内容块引用。需要 beta 头 files-api-2025-04-14。

- **UploadFile**：上传本地文件。返回文件元数据。
- **ListFiles**：列出已上传的文件（带分页）。
- **GetFile**：检索特定文件的元数据。
- **DownloadFile**：下载文件内容。

- **DeleteFile**：永久删除文件。

请求参数

Messages API 上提供的附加请求参数。

- **ServiceTier**：控制优先级层使用：'auto'（默认）或 'standard_only'。
- **InferenceGeo**：控制推理地域：'global'（默认）或 'us'。
- **Container**：用于跨请求复用代码执行环境的容器 ID。
- **IsError**：（内容块属性）在 tool_result 块上设置为 True，表示工具执行失败。
- **CacheTTL**：扩展缓存生存时间：'5m'（默认）或 '1h'。
- **ThinkingType**：设置为 'adaptive' 让 Claude 自动决定思考深度。

MCP Connector

将 Claude 连接到外部 MCP（Model Context Protocol）服务器以访问第三方工具。需要 beta 头部 `mcp-client-2025-11-20`。

- **CreateMessageWithMCP**：创建带有 MCP 服务器连接的消息。
 - **MCPServerUrl**：MCP 服务器的 HTTPS URL。
 - **MCPServerName**：服务器的唯一名称。
- **MCPServers**：（Request 属性）包含 MCP 服务器定义的数组，每个定义包含 ServerType、Url、Name 和 AuthorizationToken。
- **MCPServerName**：（Tool 属性）当 ToolType 为 'mcp_toolset' 时，引用一个 MCP 服务器。

TsgcHTTP_API_Anthropic

用于 Claude 消息、文件、批次和令牌计数的 Anthropic REST API 客户端。

简介

用于 Claude 消息、文件、批次和令牌计数的 Anthropic REST API 客户端。

简介

用于 Claude 消息、文件、批次和令牌计数的 Anthropic REST API 客户端。
使用您的 API 密钥和可选的 HTTP/重试/日志设置配置 **AnthropicOptions** 属性，然后使用请求对象调用 `CreateMessage/CreateChatCompletion` 等类型化方法并接收类型化响应类。
有关该提供商的背景信息，请参阅 [Anthropic](#) 指南。

参考

- 属性——组件上共有 5 个可用属性。
- 方法 — 7 个公共方法。
- 事件 — 可处理 2 个事件。

Anthropic | 消息

发送带有文本内容的结构化输入消息列表，模型将生成对话中的下一条消息。

简单示例

向 Claude 发送 Hello 消息。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_CreateMessage("claude-sonnet-4-20250514", "Hello!"));
```

系统提示词示例

发送带有系统提示的消息以控制 Claude 的行为。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_CreateMessageWithSystem("claude-sonnet-4-20250514",
    "You are a helpful assistant that responds in Spanish.",
    "What is the capital of France?"));
```

高级示例

使用类型化的请求/响应类对消息参数进行完全控制。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
    oRequest->Model = "claude-sonnet-4-20250514";
    oRequest->MaxTokens = 1024;
    oRequest->System = "You are a helpful assistant.";
    oRequest->Temperature = 0.7;

    TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
    oMessage->Role = "user";
    oMessage->Content = "Hello!";
    oMessages = oRequest->Messages;
    SetLength(oMessages, 1);
    oMessages[0] = oMessage;
    oRequest->Messages = oMessages;

    TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
    try {
        if (oResponse->Content.Length > 0)
            ShowMessage(oResponse->Content[0]->Text);
    } __finally {
        delete oResponse;
    }
} __finally {
    delete oMessage;
    delete oRequest;
}
```

流式示例

使用 Server-Sent Events (SSE) 实时流式传输响应。分配 OnHTTPAPISSE 事件处理程序以接收流式事件。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
Anthropic->OnHTTPAPISSE = OnSSEEvent;
Anthropic->_CreateMessageStream("claude-sonnet-4-20250514", "Tell me a story.");
```

```
void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
    const String aData, bool &Cancel)
{
    // aEvent contains the event type (e.g. content_block_delta)
    // aData contains the JSON data for this event
    Memo1->Lines->Add(aData);
}
```

Anthropic | 视觉

Claude 能够理解和分析图像。您可以在内容块中将图像作为 Base64 编码数据发送。

支持的图像格式

- **image/jpeg** - JPEG 图像
- **image/png** - PNG 图像
- **image/gif** - GIF 图像
- **image/webp** — WebP 图像

简单示例

发送带有提示的图像，要求 Claude 对其进行描述。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

// Load image and encode to base64
TFileStream *oStream = new TFileStream("photo.png", fmOpenRead);
try {
    TBytesStream *oBytes = new TBytesStream();
    try {
        oBytes->CopyFrom(oStream, 0);
        vBase64 = EncodeBase64(oBytes->Memory, oBytes->Size);
    } __finally {
        oBytes->Free();
    }
} __finally {
    oStream->Free();
}

ShowMessage(Anthropic->CreateVisionMessage("claude-sonnet-4-20250514",
    "What is in this image?", vBase64, "image/png"));
```

高级示例

使用内容块可以更好地控制图像消息。

```
TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages(NULL);
try {
    oRequest->Model = "claude-sonnet-4-20250514";
    oRequest->MaxTokens = 4096;

    TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message(NULL);
    oMessage->Role = "user";

    // Image content block
    TsgcAnthropicClass_Request_Content_Block *oImageBlock = new TsgcAnthropicClass_Request_Content_Block(NULL);
    oImageBlock->ContentType = "image";
    oImageBlock->MediaType = "image/jpeg";
    oImageBlock->Data = vBase64;

    // Text content block
    TsgcAnthropicClass_Request_Content_Block *oTextBlock = new TsgcAnthropicClass_Request_Content_Block(NULL);
    oTextBlock->ContentType = "text";
    oTextBlock->Text = "Describe this image in detail.";

    SetLength(oBlocks, 2);
    oBlocks[0] = oImageBlock;
    oBlocks[1] = oTextBlock;
    oMessage->ContentBlocks = oBlocks;

    SetLength(oMessages, 1);
    oMessages[0] = oMessage;
    oRequest->Messages = oMessages;

    TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
    try {
        if (Length(oResponse->Content) > 0)
```

```
        ShowMessage(oResponse->Content[0]->Text);  
    } __finally {  
        oResponse->Free();  
    }  
} __finally {  
    oImageBlock->Free();  
    oTextBlock->Free();  
    oMessage->Free();  
    oRequest->Free();  
}
```


Anthropic | 工具使用

Claude 支持工具使用（函数调用），允许您定义 Claude 可以在对话中调用的工具。当 Claude 决定使用某个工具时，它会返回一个 **tool_use** 内容块。然后您执行该工具，并将结果作为 **tool_result** 内容块发送回去。

工具使用流程

1. 使用名称、描述和 input_schema（JSON Schema）定义工具。
2. 发送带有工具定义的消息。
3. Claude 以 **tool_use** 内容块（stop_reason = 'tool_use'）进行响应。
4. 使用提供的输入执行工具。
5. 发送包含 **tool_result** 内容块的新消息，其中包含输出内容。
6. Claude 以最终答案作为响应。

示例

定义一个天气工具并处理工具使用循环。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

// Step 1: Create request with tools
TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages(NULL);
try {
    oRequest->Model = "claude-sonnet-4-20250514";
    oRequest->MaxTokens = 4096;

    // Define user message
    TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message(NULL);
    oMessage->Role = "user";
    oMessage->Content = "What is the weather in San Francisco?";
    SetLength(oMessages, 1);
    oMessages[0] = oMessage;
    oRequest->Messages = oMessages;

    // Define tool
    TsgcAnthropicClass_Request_Tool *oTool = new TsgcAnthropicClass_Request_Tool(NULL);
    oTool->Name = "get_weather";
    oTool->Description = "Get the current weather in a given location";
    oTool->InputSchema =
        "{\n\"type\": \"object\", \"properties\": {\n\"location\": {\n\"type\": \"string\", \"\n\"description\": \"The city and state\"}}, \"required\": [\n\"location\"]}\"";
    SetLength(oTools, 1);
    oTools[0] = oTool;
    oRequest->Tools = oTools;

    // Step 2: Send request
    TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
    try {
        // Step 3: Check if Claude wants to use a tool
        if (oResponse->StopReason == "tool_use") {
            // Find the tool_use content block
            for (int i = 0; i < Length(oResponse->Content); i++) {
                if (oResponse->Content[i]->ContentType == "tool_use") {
                    vToolUseId = oResponse->Content[i]->Id;
                    vToolName = oResponse->Content[i]->Name;
                    vToolInput = oResponse->Content[i]->Input;
                    // Step 4: Execute your tool (get_weather) and get result
                    vToolResult = "72 degrees and sunny";
                    break;
                }
            }

            // Step 5: Send tool result back
            // Build new message array with assistant response + tool result
            // ... (continue the conversation with tool_result content block)
        }
    } finally {
        oResponse->Free();
    }
} finally {
    oMessage->Free();
    oTool->Free();
}
```

```
oRequest->Free();  
}
```

Anthropic | Models

列出并检索有关可用 Claude 模型的信息。

列出模型

列出所有可用的 Claude 模型。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_GetModels());
```

获取模型

检索有关特定模型的信息。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_GetModel("claude-sonnet-4-20250514"));
```

类型化响应

使用带类型的响应类对模型数据进行结构化访问。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Response_Models *oModels = Anthropic->GetModels();
try {
    for (int i = 0; i < oModels->Data.Length; i++)
        ShowMessage(oModels->Data[i]->Id + " - " + oModels->Data[i]->DisplayName);
} __finally {
    delete oModels;
}
```

Anthropic | 消息批处理

消息批处理 API 允许您异步处理大量消息。这非常适合不需要即时响应的任务，例如批量内容生成、数据分析或批处理工作流。

列出批处理任务

列出您组织的所有消息批次。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_ListBatches());
```

获取批次

检索有关特定批次的信息。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_GetBatch("batch_id_here"));
```

取消批处理

取消仍在处理中的批次。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_CancelBatch("batch_id_here"));
```

获取批处理结果

检索已完成批次的结果。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_GetBatchResults("batch_id_here"));
```

类型化响应

使用类型化响应类对批量数据进行结构化访问。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Response_Batches *oBatches = Anthropic->ListBatches();
try {
    for (int i = 0; i < Length(oBatches->Batches); i++)
        ShowMessage(oBatches->Batches[i]->Id + " - " + oBatches->Batches[i]->ProcessingStatus);
} __finally {
    oBatches->Free();
}
```

Anthropic | 扩展思考

扩展思考使 Claude 能够在提供响应之前逐步思考复杂问题。启用后，Claude 会创建内部推理（思考块），从而提高数学、编码、分析和其他复杂任务的响应质量。

简单示例

使用便捷方法发送启用了扩展思考的消息。温度参数自动设置为 1.0（启用思考时 API 要求如此）。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_CreateMessageWithThinking("claude-sonnet-4-20250514",
    "How many r's are in the word strawberry?", 10000));
```

高级示例

使用带类型的请求/响应类可实现完全控制。将 ThinkingType 设为 'enabled'，并将 ThinkingBudgetTokens 设为所需的令牌预算（最少 1024）。响应将包含思考内容块和文本内容块。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
    oRequest->Model = "claude-sonnet-4-20250514";
    oRequest->MaxTokens = 16384;
    oRequest->ThinkingType = "enabled";
    oRequest->ThinkingBudgetTokens = 10000;

    TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
    oMessage->Role = "user";
    oMessage->Content = "Explain the proof that there are infinitely many primes.";
    oMessages = oRequest->Messages;
    SetLength(oMessages, 1);
    oMessages[0] = oMessage;
    oRequest->Messages = oMessages;

    TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
    try {
        for (int i = 0; i < oResponse->Content.Length; i++) {
            if (oResponse->Content[i]->ContentType == "thinking")
                ShowMessage("Thinking: " + oResponse->Content[i]->Thinking);
            else if (oResponse->Content[i]->ContentType == "text")
                ShowMessage("Response: " + oResponse->Content[i]->Text);
        }
    } __finally {
        delete oResponse;
    }
} __finally {
    delete oMessage;
    delete oRequest;
}
```

带思考的多轮对话

在多轮对话中使用扩展思维时，之前响应中的 thinking 和 redacted_thinking 块必须在对话中传回。使用 ContentBlocks 数组包含这些块。

```
// Pass back thinking blocks from a previous response
TsgcAnthropicClass_Request_Content_Block *oBlock = new TsgcAnthropicClass_Request_Content_Block();
oBlock->ContentType = "thinking";
oBlock->Text = oPrevThinkingBlock->Thinking; // thinking text
oBlock->Signature = oPrevThinkingBlock->Signature; // signature string

// Pass back redacted thinking blocks
TsgcAnthropicClass_Request_Content_Block *oBlock2 = new TsgcAnthropicClass_Request_Content_Block();
oBlock2->ContentType = "redacted_thinking";
```

```
oBlock2->Data = oPrevRedactedBlock->Data;
```

属性

- **ThinkingType**：设为 'enabled' 可激活扩展思维，设为 'disabled' 可将其关闭。
- **ThinkingBudgetTokens**：思考过程的最大令牌数（最小 1024）。必须小于 MaxTokens。

注意事项

- 启用 thinking 时，温度必须为 1.0（便捷方法会自动完成此设置）。
- 当启用 thinking 时，不能使用 TopK。
- BudgetTokens 必须至少为 1024 且小于 MaxTokens。
- 响应内容块可能包括：thinking、redacted_thinking 和 text 类型。

Anthropic | 文档

Claude 可以处理作为内容块在消息中发送的 PDF 文档和文本文件。文档以 base64 编码数据或通过 URL 发送，Claude 可以分析、总结并回答有关其内容的问题。

简单示例

使用便捷方法发送带有问题的 PDF 文档。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

// Load PDF file and encode to base64
AnsiString vBase64 = sgcBase64Encode(LoadFileToBytes("document.pdf"));
ShowMessage(Anthropic->CreateDocumentMessage("claude-sonnet-4-20250514",
    "Summarize this document.", vBase64, "application/pdf"));
```

带引用的高级示例

使用类型化类发送启用引用的文档。启用引用后，Claude 的回复将包含对源文档特定部分的引用。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
    oRequest->Model = "claude-sonnet-4-20250514";
    oRequest->MaxTokens = 4096;

    // Create document content block
    TsgcAnthropicClass_Request_Content_Block *oDocBlock = new TsgcAnthropicClass_Request_Content_Block();
    oDocBlock->ContentType = "document";
    oDocBlock->SourceType = "base64";
    oDocBlock->MediaType = "application/pdf";
    oDocBlock->Data = sgcBase64Encode(LoadFileToBytes("report.pdf"));
    oDocBlock->Title = "Annual Report";
    oDocBlock->CitationsEnabled = true;

    // Create text prompt block
    TsgcAnthropicClass_Request_Content_Block *oTextBlock = new TsgcAnthropicClass_Request_Content_Block();
    oTextBlock->ContentType = "text";
    oTextBlock->Text = "What are the key findings?";

    // Build message with content blocks
    TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
    oMessage->Role = "user";
    oBlocks = oMessage->ContentBlocks;
    SetLength(oBlocks, 2);
    oBlocks[0] = oDocBlock;
    oBlocks[1] = oTextBlock;
    oMessage->ContentBlocks = oBlocks;

    oMessages = oRequest->Messages;
    SetLength(oMessages, 1);
    oMessages[0] = oMessage;
    oRequest->Messages = oMessages;

    TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
    try {
        if (oResponse->Content.Length > 0)
            ShowMessage(oResponse->Content[0]->Text);
    } __finally {
        delete oResponse;
    }
} __finally {
    delete oRequest;
}
```

属性

- **ContentType**：对于文档内容块，设置为 'document'。
- **SourceType**：源类型：'base64'（默认）、'text' 或 'url'。
- **MediaType**：MIME 类型（例如 application/pdf、text/plain）。
- **Data**：base64 编码的文档数据，或当 SourceType 为 'url' 时的 URL。
- **Title**：可选的文档标题，供参考使用。
- **Context**：关于文档的可选上下文或元数据。
- **CitationsEnabled**：设置为 True 以在响应中启用来源引用。

支持的格式

- **PDF**：application/pdf（最大 32 MB，每次请求最多 100 页）
- **纯文本**：text/plain

Anthropic | 提示词缓存

提示缓存允许您在 API 调用之间缓存常用上下文，在缓存读取时最多可降低 90% 的成本并改善延迟。缓存内容通过 `cache_control` 参数标记，并在缓存 TTL 窗口内跨请求复用。

系统提示缓存

缓存长系统提示以避免每次请求都重新处理。将 `SystemCacheControl` 设置为 `True` 可自动用 `cache_control` 包装系统提示。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages(NULL);
try {
    oRequest->Model = "claude-sonnet-4-20250514";
    oRequest->MaxTokens = 4096;
    oRequest->System = "You are an expert legal assistant... (long system prompt)";
    oRequest->SystemCacheControl = true; // Enable caching for system prompt

    TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message(NULL);
    oMessage->Role = "user";
    oMessage->Content = "Analyze this contract clause.";
    oMessages = oRequest->Messages;
    SetLength(oMessages, 1);
    oMessages[0] = oMessage;
    oRequest->Messages = oMessages;

    TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
    try {
        // Check cache usage in response
        ShowMessage("Cache created: " +
            IntToStr(oResponse->Usage->CacheCreationInputTokens));
        ShowMessage("Cache read: " +
            IntToStr(oResponse->Usage->CacheReadInputTokens));
        ShowMessage(oResponse->Content[0]->Text);
    } __finally {
        oResponse->Free();
    }
} __finally {
    oMessage->Free();
    oRequest->Free();
}
```

内容块缓存

通过在单个内容块上将 `CacheControl` 设置为 'ephemeral'，可以缓存特定内容块（例如大型文档或上下文）。

```
TsgcAnthropicClass_Request_Content_Block *oBlock = new TsgcAnthropicClass_Request_Content_Block(NULL);
oBlock->ContentType = "text";
oBlock->Text = "(large reference text to cache)";
oBlock->CacheControl = "ephemeral"; // Mark for caching
```

工具定义缓存

缓存工具定义，避免在每次请求时重新处理。当您有许多在请求间保持不变的工具定义时，此功能非常有用。

```
TsgcAnthropicClass_Request_Tool *oTool = new TsgcAnthropicClass_Request_Tool(NULL);
oTool->Name = "search_database";
oTool->Description = "Search the database for records.";
oTool->InputSchema = "{\"type\":\"object\",\"properties\":{\"query\":{\"type\":\"string\"}}\"";
oTool->CacheControl = "ephemeral"; // Cache this tool definition
```

属性

- **SystemCacheControl**：在请求中设置为 True 可使用临时 cache_control 缓存系统提示词。
- **CacheControl**：在内容块或工具定义上设置为 'ephemeral' 以将其标记为缓存。
- **CacheCreationInputTokens**：（响应）用于创建缓存条目的令牌数量。
- **CacheReadInputTokens**：（响应）从缓存中读取的令牌数量（节省费用）。

定价

- **缓存写入**：基本输入令牌价格的 1.25 倍（TTL 为 5 分钟）。
- **缓存读取**：基础输入令牌价格的 0.1 倍（节省 90%）。
- **Cache TTL**：默认 5 分钟。后续请求将刷新 TTL。

Anthropic | 引用

当启用引用功能发送文档时，Claude 的响应会包含对源文档特定部分的回溯引用。这让您验证声明并追溯信息来源。

启用引用

在文档内容块上将 CitationsEnabled 设置为 True。引用必须在请求中的所有文档上全部启用或全部禁用。

```
TsgcAnthropicClass_Request_Content_Block *oDocBlock = new TsgcAnthropicClass_Request_Content_Block(NULL);
oDocBlock->ContentType = "document";
oDocBlock->SourceType = "base64";
oDocBlock->MediaType = "application/pdf";
oDocBlock->Data = sgcBase64Encode(LoadFileToBytes("report.pdf"));
oDocBlock->CitationsEnabled = true; // Enable citations
```

从响应中读取引用

响应中的文本内容块可能包含带有源引用的 Citations 数组。

```
TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
try {
    for (int i = 0; i < Length(oResponse->Content); i++) {
        if (oResponse->Content[i]->ContentType == "text") {
            ShowMessage(oResponse->Content[i]->Text);

            // Process citations
            for (int j = 0; j < Length(oResponse->Content[i]->Citations); j++) {
                TsgcAnthropicClass_Response_Citation *oCitation = oResponse->Content[i]->Citations[j];
                ShowMessage(Format(" Citation [%s]: \"%s\"",
                    ARRAYOFCNST((oCitation->CitationType, oCitation->CitedText))));

                if (oCitation->CitationType == "page_location")
                    ShowMessage(Format(" Pages %d-%d of \"%s\"",
                        ARRAYOFCNST((oCitation->StartPageNumber, oCitation->EndPageNumber,
                            oCitation->DocumentTitle))));
            }
        }
    }
} __finally {
    oResponse->Free();
}
```

引文类型

- **char_location**：纯文本文档中的字符级引用。字段：DocumentIndex、DocumentTitle、StartCharIndex、EndCharIndex。
- **page_location**：PDF 文档中的页面级引用。字段：DocumentIndex、DocumentTitle、StartPageNumber、EndPageNumber。
- **content_block_location**：自定义内容文档中的块级引用。字段：DocumentIndex、DocumentTitle、StartBlockIndex、EndBlockIndex。
- **web_search_result_location**：网络搜索结果的引用。字段：Url、Title、EncryptedIndex。

Anthropic | 网络搜索

Web 搜索工具使 Claude 能够在对话中实时搜索 Web 信息。这是一个由 Anthropic 托管并自动执行的内置服务器端工具。

简单示例

使用启用 Web 搜索的便捷方法创建消息。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
ShowMessage(Anthropic->_CreateMessageWithWebSearch("claude-sonnet-4-20250514",
    "What are the latest news about Delphi programming?"));
```

高级示例

使用带类型的类可完全控制网络搜索工具参数，例如 MaxUses。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
    oRequest->Model = "claude-sonnet-4-20250514";
    oRequest->MaxTokens = 4096;

    // Add web search tool
    TsgcAnthropicClass_Request_Tool *oTool = new TsgcAnthropicClass_Request_Tool();
    oTool->ToolType = "web_search_20250305";
    oTool->Name = "web_search";
    oTool->MaxUses = 5; // Max 5 searches per request
    oTools = oRequest->Tools;
    SetLength(oTools, 1);
    oTools[0] = oTool;
    oRequest->Tools = oTools;

    TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
    oMessage->Role = "user";
    oMessage->Content = "Find the current price of Bitcoin.";
    oMessages = oRequest->Messages;
    SetLength(oMessages, 1);
    oMessages[0] = oMessage;
    oRequest->Messages = oMessages;

    TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
    try {
        for (int i = 0; i < oResponse->Content.Length; i++) {
            if (oResponse->Content[i]->ContentType == "text")
                ShowMessage(oResponse->Content[i]->Text);
        }
    } finally {
        delete oResponse;
    }
} finally {
    delete oTool;
    delete oMessage;
    delete oRequest;
}
```

内置工具类型

- **web_search_20250305**：服务器端 Web 搜索。属性：Name ('web_search')、MaxUses。
- **code_execution_20250825**：沙盒代码执行。属性：Name ('code_execution')、MaxUses。
- **computer_20250124**：计算机使用（鼠标、键盘、截图）。属性：Name ('computer')、DisplayWidthPx、DisplayHeightPx。

响应内容类型

使用内置工具时，响应可能包含额外的内容块类型：

- **server_tool_use**：表示 Claude 调用了内置工具。字段：Id、Name、Input。
- **web_search_tool_result**：包含网络搜索结果。字段：ToolUseId、Data（原始 JSON 内容）。

Anthropic | Structured Outputs

结构化输出强制 Claude 返回符合 JSON 架构的响应。这保证了与您的架构定义匹配的有效可解析 JSON 输出。

简单示例

使用便捷方法创建带有 JSON 架构输出的消息。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

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

ShowMessage(Anthropic->CreateMessageJSON("claude-sonnet-4-20250514",
    "Extract the name and age: John is 30 years old.", vSchema));
```

高级示例

使用类型化类将结构化输出与 effort 参数结合使用。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
    oRequest->Model = "claude-sonnet-4-20250514";
    oRequest->MaxTokens = 4096;

    // JSON schema for structured output
    oRequest->OutputFormatSchema =
        "{\"type\":\"object\",\"properties\":{\"sentiment\":{\"type\":\"string\", \"enum\": [\"positive\", \"negative\", \"neutral\"]}, \"confidence\":{\"type\":\"number\"}}, \"required\": [\"sentiment\", \"confidence\"], \"additionalProperties\": false}";

    // Set effort level (low, medium, high, max)
    oRequest->Effort = "medium";

    TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
    oMessage->Role = "user";
    oMessage->Content = "Analyze the sentiment: I love this product!";
    oMessages = oRequest->Messages;
    SetLength(oMessages, 1);
    oMessages[0] = oMessage;
    oRequest->Messages = oMessages;

    TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
    try {
        for (int i = 0; i < oResponse->Content.Length; i++) {
            if (oResponse->Content[i]->ContentType == "text")
                ShowMessage(oResponse->Content[i]->Text);
        }
    } __finally {
        delete oResponse;
    }
} __finally {
    delete oMessage;
    delete oRequest;
}
```

严格工具使用

在工具定义上启用严格模式，以确保工具输入严格符合 input_schema。

```
TsgcAnthropicClass_Request_Tool *oTool = new TsgcAnthropicClass_Request_Tool();
oTool->Name = "get_weather";
oTool->Description = "Get the current weather for a location";
```

```
oTool->Strict = true; // Guarantee schema conformance
oTool->InputSchema = "{\"type\":\"object\",\"properties\":{\"location\":\""
    "\"type\":\"string\"}},\"required\":[\"location\"],\"additionalProperties\":false}";
```

属性

- **OutputFormatSchema** : JSON Schema 字符串。Claude 的响应文本将是符合此模式的有效 JSON。
- **Effort** : 控制输出质量与成本。值：'low'、'medium'、'high'（默认）、'max'（仅限 Opus 4.6）。
- **Strict** : （针对工具）当为 True 时，工具输入保证符合 input_schema。

Anthropic | Files API

Files API 允许您上传、列出、检索、下载和删除文件。已上传的文件可以通过文件源类型的文档内容块在消息中引用。

注意：Files API 需要 beta 请求头 `files-api-2025-04-14`。请在 **`AnthropicOptions.BetaHeaders`** 属性中设置此请求头。

上传文件

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
Anthropic->AnthropicOptions->BetaHeaders = "files-api-2025-04-14";

TsgcAnthropicClass_Response_File *oFile = Anthropic->UploadFile("C:\\documents\\report.pdf");
try {
    ShowMessage("File ID: " + oFile->Id);
} __finally {
    delete oFile;
}
```

列出文件

```
TsgcAnthropicClass_Response_FileList *oList = Anthropic->ListFiles(50);
try {
    for (int i = 0; i < oList->Data.Length; i++)
        ShowMessage(oList->Data[i]->Id + " - " + oList->Data[i]->Filename +
            " (" + IntToStr(oList->Data[i]->SizeBytes) + " bytes)");
    ShowMessage("Has more: " + BoolToStr(oList->HasMore, true));
} __finally {
    delete oList;
}
```

在消息中使用文件

使用将 `SourceType` 设置为 'file' 的文档内容块来引用已上传的文件。

```
TsgcAnthropicClass_Request_Content_Block *oDocBlock = new TsgcAnthropicClass_Request_Content_Block();
oDocBlock->ContentType = "document";
oDocBlock->SourceType = "file";
oDocBlock->FileId = "file_abc123"; // ID from UploadFile

TsgcAnthropicClass_Request_Content_Block *oTextBlock = new TsgcAnthropicClass_Request_Content_Block();
oTextBlock->ContentType = "text";
oTextBlock->Text = "Summarize this document.";
```

删除文件

```
TsgcAnthropicClass_Response_File *oDeleted = Anthropic->DeleteFile("file_abc123");
try {
    ShowMessage("Deleted: " + oDeleted->Id);
} __finally {
    delete oDeleted;
}
```

API 方法

- **UploadFile**：从本地路径上传文件。返回文件元数据（Id、Filename、MimeType、SizeBytes）。
- **ListFiles**：列出所有已上传的文件，支持分页（Limit、AfterId、BeforeId）。
- **GetFile**：通过 ID 检索特定文件的元数据。
- **DownloadFile**：下载文件内容（仅限代码执行工具创建的文件可下载）。

- **DeleteFile**：通过 ID 永久删除文件。

Anthropic | MCP 连接器

MCP（模型上下文协议）连接器允许 Claude 访问外部 MCP 服务器上的工具。这使得与第三方服务和自定义工具提供商的集成成为可能。

注意：MCP 连接器需要 Beta 标头 `mcp-client-2025-11-20`。请在 **`AnthropicOptions.BetaHeaders`** 属性中进行设置。

简单示例

使用便捷方法通过 MCP 服务器创建消息。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
Anthropic->AnthropicOptions->BetaHeaders = "mcp-client-2025-11-20";

ShowMessage(Anthropic->CreateMessageWithMCP("claude-sonnet-4-20250514",
    "What tools are available?",
    "https://my-mcp-server.example.com/sse",
    "my-mcp-server"));
```

高级示例

使用类型化类可以完全控制 MCP 服务器配置，包括认证。

```
TsgcHTTP_API_Anthropic *Anthropic = new TsgcHTTP_API_Anthropic(NULL);
Anthropic->AnthropicOptions->ApiKey = "API_KEY";
Anthropic->AnthropicOptions->BetaHeaders = "mcp-client-2025-11-20";

TsgcAnthropicClass_Request_Messages *oRequest = new TsgcAnthropicClass_Request_Messages();
try {
    oRequest->Model = "claude-sonnet-4-20250514";
    oRequest->MaxTokens = 4096;

    // Configure MCP server
    TsgcAnthropicClass_Request_MCPServer *oServer = new TsgcAnthropicClass_Request_MCPServer();
    oServer->ServerType = "url";
    oServer->Url = "https://my-mcp-server.example.com/sse";
    oServer->Name = "my-server";
    oServer->AuthorizationToken = "OAUTH_TOKEN"; // Optional auth
    oServers = oRequest->MCPServers;
    SetLength(oServers, 1);
    oServers[0] = oServer;
    oRequest->MCPServers = oServers;

    // Add MCP toolset
    TsgcAnthropicClass_Request_Tool *oTool = new TsgcAnthropicClass_Request_Tool();
    oTool->ToolType = "mcp_toolset";
    oTool->MCPServerName = "my-server";
    oTools = oRequest->Tools;
    SetLength(oTools, 1);
    oTools[0] = oTool;
    oRequest->Tools = oTools;

    // Add user message
    TsgcAnthropicClass_Request_Message *oMessage = new TsgcAnthropicClass_Request_Message();
    oMessage->Role = "user";
    oMessage->Content = "Search for recent news about AI.";
    oMessages = oRequest->Messages;
    SetLength(oMessages, 1);
    oMessages[0] = oMessage;
    oRequest->Messages = oMessages;

    TsgcAnthropicClass_Response_Messages *oResponse = Anthropic->CreateMessage(oRequest);
    try {
        for (int i = 0; i < oResponse->Content.Length; i++) {
            if (oResponse->Content[i]->ContentType == "text")
                ShowMessage(oResponse->Content[i]->Text);
            else if (oResponse->Content[i]->ContentType == "mcp_tool_use")
                ShowMessage("MCP tool call: " + oResponse->Content[i]->Name +
                    " on " + oResponse->Content[i]->ServerName);
        }
    }
} finally {
```

```
    delete oResponse;
  }
} __finally {
  delete oServer;
  delete oTool;
  delete oMessage;
  delete oRequest;
}
```

MCP 服务器属性

- **ServerType**：目前仅支持 'url'。
- **Url**：MCP 服务器的 HTTPS URL。
- **Name**：此服务器的唯一标识符。必须与工具集中的 MCPServerName 匹配。
- **AuthorizationToken**：用于已认证服务器的可选 OAuth Bearer 令牌。

响应内容类型

- **mcp_tool_use**：Claude 调用了 MCP 工具。字段：Id、Name、ServerName、Input。
- **mcp_tool_result**：MCP 工具执行的结果。字段：ToolUseId、IsError、Data（内容）。

Gemini

Google Gemini 是由 Google DeepMind 开发的多模态 AI 模型系列。Gemini 模型支持文本生成、视觉、结构化输出、嵌入和工具使用，为构建 AI 驱动的应用程序提供了强大的能力。
sgcWebSockets 库提供了 Delphi 组件 **TsgcHTTP_API_Gemini**，用于与 Gemini API 交互。

Gemini API

Gemini API 提供对 Google Gemini 模型的访问，用于构建 AI 驱动的应用程序。该 API 支持内容生成、视觉（图像理解）、结构化 JSON 输出、流式传输、令牌计数、嵌入向量、工具使用（函数调用）和模型列表。

功能特性

- 消息
 - [Gemini 消息示例](#)
- 视觉
 - [Gemini 视觉示例](#)
- 模型
 - [Gemini 模型示例](#)
- 结构化输出
 - [Gemini 结构化输出示例](#)
- 令牌计数
 - [Gemini 令牌计数示例](#)
- 嵌入
 - [Gemini 嵌入示例](#)
- 工具使用
 - [Gemini 工具使用示例](#)

配置

Gemini API 使用 API 密钥进行身份验证。请访问 Google AI Studio 中的 [API 密钥](#) 页面，以检索您在请求中使用的 API 密钥。

请记住，您的 API 密钥是机密！请勿与他人共享或在任何客户端代码中公开它。

此 **API Key** 必须在组件的 **GeminiOptions.ApiKey** 属性中配置。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "YOUR_API_KEY";
```

消息

向 Gemini 模型发送内容并接收生成的响应。模型根据提供的输入生成下一条消息。

- **_CreateContent**：使用指定的模型和用户提示创建内容。
 - **Model**：要使用的模型（例如 gemini-2.0-flash）。
 - **Message**：用户消息内容。
 - **MaxOutputTokens**：要生成的最大令牌数。
- **_CreateContentWithSystem**：使用系统指令创建内容。
 - **Model**：要使用的模型。
 - **System**：设置模型行为的系统指令。
 - **Message**：用户消息内容。
 - **MaxOutputTokens**：要生成的最大令牌数。
- **_CreateContentStream**：创建启用流式传输 (SSE) 的内容。事件通过 OnHTTPAPISSE 事件处理程序传递。

视觉

Gemini 模型可以理解以 Base64 编码内容形式传递的图像以及文本提示。

- **_CreateVisionContent**：发送带有文本提示的图像。
 - **Model**：要使用的模型。
 - **Prompt**：附带图像的文本提示。
 - **Base64**：base64 编码的图像数据。
 - **MediaType**：MIME 类型（image/jpeg、image/png、image/gif、image/webp）。
 - **MaxOutputTokens**：生成的最大令牌数。

结构化输出

通过提供定义预期响应格式的 JSON 模式，从 Gemini 模型生成结构化 JSON 输出。

- **_CreateContentJSON**：创建带有结构化 JSON 输出的内容。
 - **Model**：要使用的模型。
 - **Message**：用户消息内容。
 - **Schema**：定义预期输出结构的 JSON Schema。
 - **MaxOutputTokens**：要生成的最大令牌数。

模型

列出并检索可用 Gemini 模型的详细信息。

- **_GetModels**：列出所有可用模型。
- **_GetModel**：获取特定模型的详细信息。
 - **ModelId**：要检索的模型标识符。

令牌计数

在将消息发送给模型之前统计其令牌数量。

- **_CountTokens**：计算消息的令牌数。
 - **Model**：用于令牌化的模型。
 - **Message**：要计算令牌的文本内容。

嵌入向量

使用 Gemini 模型为文本内容生成向量嵌入。

- **_EmbedContent**：为文本生成嵌入向量。
 - **Model**：用于生成嵌入向量的模型。
 - **Text**：需要生成嵌入向量的文本内容。

TsgcHTTP_API_Gemini

Google Gemini REST API 客户端，用于内容生成、令牌计数和嵌入。

简介

Google Gemini REST API 客户端，用于内容生成、令牌计数和嵌入。

简介

Google Gemini REST API 客户端，用于内容生成、令牌计数和嵌入。
使用您的 API 密钥和可选的 HTTP/重试/日志设置配置 **GeminiOptions** 属性，然后使用请求对象调用 `CreateMessage/`
`CreateChatCompletion` 等类型化方法，并接收类型化响应类。
有关此提供者的背景信息，请参阅 [Gemini](#) 指南。

参考

- 属性 — 组件上提供 5 个属性。
- 方法 — 3 个公共方法。
- 事件——可处理 2 个事件。

DeepSeek

DeepSeek 是一家专注于构建强大开源语言模型的中国 AI 公司。其模型在编程、推理和通用任务方面表现出色，以具有竞争力的价格提供强劲性能。

sgcWebSockets 库提供 Delphi 组件 **TsgcHTTP_API_DeepSeek**，用于与 DeepSeek API 交互。

DeepSeek API

DeepSeek API 提供对 DeepSeek 模型的访问，用于构建 AI 驱动的应用程序。该 API 支持文本生成、视觉（图像理解）、流式传输和模型列表。该 API 遵循 OpenAI 兼容格式，易于集成。

功能特性

- 消息
 - [DeepSeek 消息示例](#)
- Vision
 - [DeepSeek Vision 示例](#)
- 模型
 - [DeepSeek 模型示例](#)

配置

DeepSeek API 使用 API 密钥进行身份验证。请访问 DeepSeek 平台的 [API 密钥](#) 页面，获取您将在请求中使用的 API 密钥。

请记住，您的 API 密钥是机密！请勿与他人共享或在任何客户端代码中公开它。

此 **API Key** 必须在组件的 **DeepSeekOptions.ApiKey** 属性中配置。

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "YOUR_API_KEY";
```

消息

发送带有文本内容的结构化输入消息列表，模型将生成对话中的下一条消息。

- **_CreateMessage**：使用指定的模型和用户提示创建消息。
 - **Model**：要使用的模型（例如 deepseek-chat）。
 - **Message**：用户消息内容。
- **_CreateMessageWithSystem**：创建带有系统提示词的消息。
 - **System**：设置助手行为的系统提示词。
- **_CreateMessageStream**：创建启用流式传输（SSE）的消息。事件通过 OnHTTPAPISSE 事件处理程序传递。

视觉

DeepSeek 能够理解消息中以 base64 编码内容传递的图像。

- **_CreateVisionMessage**：发送带有文本提示的图像。
 - **ImageBase64**：Base64 编码的图像数据。
 - **MediaType**：MIME 类型（image/jpeg、image/png、image/gif、image/webp）。
 - **Prompt**：随图像发送的文本提示。

模型

列出可用的 DeepSeek 模型。

- **_GetModels**：列出所有可用模型。

TsgcHTTP_API_DeepSeek

用于聊天补全的 DeepSeek REST API 客户端。

简介

用于聊天补全的 DeepSeek REST API 客户端。

简介

用于聊天补全的 DeepSeek REST API 客户端。

使用您的 API 密钥和可选的 HTTP/重试/日志设置配置 **DeepSeekOptions** 属性，然后使用请求对象调用如 `CreateMessage/CreateChatCompletion` 等类型化方法，并接收类型化响应类。
有关该提供商的背景信息，请参阅 [DeepSeek](#) 指南。

参考

- 属性——组件上提供 5 个属性。
- 方法 — 1 个公共方法。
- Events — 2 个可处理的事件。

DeepSeek | 消息

发送带有文本内容的结构化输入消息列表，模型将生成对话中的下一条消息。

简单示例

向 DeepSeek 发送 Hello 消息。

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "API_KEY";
ShowMessage(DeepSeek->_CreateMessage("deepseek-chat", "Hello!"));
```

系统提示词示例

发送带有系统提示的消息以控制 DeepSeek 的行为。

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "API_KEY";
ShowMessage(DeepSeek->_CreateMessageWithSystem("deepseek-chat",
    "You are a helpful assistant that responds in Spanish.",
    "What is the capital of France?"));
```

流式示例

使用 Server-Sent Events (SSE) 实时流式传输响应。分配 OnHTTAPISSE 事件处理程序以接收流式事件。

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "API_KEY";
DeepSeek->OnHTTAPISSE = OnSSEEvent;
DeepSeek->_CreateMessageStream("deepseek-chat", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
    const String aData, bool &Cancel)
{
    // aEvent contains the event type
    // aData contains the JSON data for this event
    Memo1->Lines->Add(aData);
}
```

DeepSeek | Vision

DeepSeek 能够理解和分析图像。您可以在内容块中以 base64 编码数据的形式发送图像。

视觉示例

向 DeepSeek 发送带提示词的图像以请求描述。

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);
DeepSeek->DeepSeekOptions->ApiKey = "API_KEY";

// Load image and encode to base64
TFileStream *oStream = new TFileStream("photo.jpg", fmOpenRead);
try {
    TBytesStream *oBytes = new TBytesStream();
    try {
        oBytes->CopyFrom(oStream, 0);
        vImageBase64 = EncodeBase64(oBytes->Memory, oBytes->Size);
    } __finally {
        oBytes->Free();
    }
} __finally {
    oStream->Free();
}

ShowMessage(DeepSeek->_CreateVisionMessage("deepseek-chat",
    "Describe this image", vImageBase64, "image/jpeg"));
```

DeepSeek | 模型

列出可用的 DeepSeek 模型。

列出模型

列出所有可用的 DeepSeek 模型。

```
TsgcHTTP_API_DeepSeek *DeepSeek = new TsgcHTTP_API_DeepSeek(NULL);  
DeepSeek->DeepSeekOptions->ApiKey = "API_KEY";  
ShowMessage(DeepSeek->_GetModels());
```

Gemini

Google Gemini 是由 Google DeepMind 开发的多模态 AI 模型系列。Gemini 模型支持文本生成、视觉、结构化输出、嵌入和工具使用，为构建 AI 驱动的应用程序提供了强大的能力。
sgcWebSockets 库提供了 Delphi 组件 **TsgcHTTP_API_Gemini**，用于与 Gemini API 交互。

Gemini API

Gemini API 提供对 Google Gemini 模型的访问，用于构建 AI 驱动的应用程序。该 API 支持内容生成、视觉（图像理解）、结构化 JSON 输出、流式传输、令牌计数、嵌入向量、工具使用（函数调用）和模型列表。

功能特性

- 消息
 - [Gemini 消息示例](#)
- 视觉
 - [Gemini 视觉示例](#)
- 模型
 - [Gemini 模型示例](#)
- 结构化输出
 - [Gemini 结构化输出示例](#)
- 令牌计数
 - [Gemini 令牌计数示例](#)
- 嵌入
 - [Gemini 嵌入示例](#)
- 工具使用
 - [Gemini 工具使用示例](#)

配置

Gemini API 使用 API 密钥进行身份验证。请访问 Google AI Studio 中的 [API 密钥](#) 页面，以检索您在请求中使用的 API 密钥。

请记住，您的 API 密钥是机密！请勿与他人共享或在任何客户端代码中公开它。

此 **API Key** 必须在组件的 **GeminiOptions.ApiKey** 属性中配置。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "YOUR_API_KEY";
```

消息

向 Gemini 模型发送内容并接收生成的响应。模型根据提供的输入生成下一条消息。

- **_CreateContent**：使用指定的模型和用户提示创建内容。
 - **Model**：要使用的模型（例如 gemini-2.0-flash）。
 - **Message**：用户消息内容。
 - **MaxOutputTokens**：要生成的最大令牌数。
- **_CreateContentWithSystem**：使用系统指令创建内容。
 - **Model**：要使用的模型。
 - **System**：设置模型行为的系统指令。
 - **Message**：用户消息内容。
 - **MaxOutputTokens**：要生成的最大令牌数。
- **_CreateContentStream**：创建启用流式传输 (SSE) 的内容。事件通过 OnHTTPAPISSE 事件处理程序传递。

视觉

Gemini 模型可以理解以 Base64 编码内容形式传递的图像以及文本提示。

- **_CreateVisionContent**：发送带有文本提示的图像。
 - **Model**：要使用的模型。
 - **Prompt**：附带图像的文本提示。
 - **Base64**：base64 编码的图像数据。
 - **MediaType**：MIME 类型（image/jpeg、image/png、image/gif、image/webp）。
 - **MaxOutputTokens**：生成的最大令牌数。

结构化输出

通过提供定义预期响应格式的 JSON 模式，从 Gemini 模型生成结构化 JSON 输出。

- **_CreateContentJSON**：创建带有结构化 JSON 输出的内容。
 - **Model**：要使用的模型。
 - **Message**：用户消息内容。
 - **Schema**：定义预期输出结构的 JSON Schema。
 - **MaxOutputTokens**：要生成的最大令牌数。

模型

列出并检索可用 Gemini 模型的详细信息。

- **_GetModels**：列出所有可用模型。
- **_GetModel**：获取特定模型的详细信息。
 - **ModelId**：要检索的模型标识符。

令牌计数

在将消息发送给模型之前统计其令牌数量。

- **_CountTokens**：计算消息的令牌数。
 - **Model**：用于令牌化的模型。
 - **Message**：要计算令牌的文本内容。

嵌入向量

使用 Gemini 模型为文本内容生成向量嵌入。

- **_EmbedContent**：为文本生成嵌入向量。
 - **Model**：用于生成嵌入向量的模型。
 - **Text**：需要生成嵌入向量的文本内容。

Gemini | 消息

向 Gemini 模型发送提示词并接收生成的内容。Gemini API 使用 generateContent 端点处理文本输入并返回模型响应。

简单示例

向 Gemini 发送 Hello 消息。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_CreateContent("gemini-2.0-flash", "Hello!"));
```

系统指令示例

发送带有系统指令的消息以控制 Gemini 的行为。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_CreateContentWithSystem("gemini-2.0-flash",
    "You are a helpful assistant that responds in Spanish.",
    "What is the capital of France?"));
```

流式示例

使用 Server-Sent Events (SSE) 实时流式传输响应。分配 OnHTTPAPISSE 事件处理程序以接收流式事件。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
Gemini->OnHTTPAPISSE = OnSSEEvent;
Gemini->_CreateContentStream("gemini-2.0-flash", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
    const String aData, bool &Cancel)
{
    Memo1->Lines->Add(aData);
}
```

高级示例

使用类型化的请求/响应类完全控制内容生成参数。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

TsgcGeminiClass_Request_GenerateContent *oRequest = new TsgcGeminiClass_Request_GenerateContent();
try {
    oRequest->Model = "gemini-2.0-flash";
    oRequest->MaxOutputTokens = 4096;
    oRequest->Temperature = 0.7;
    oRequest->SystemInstruction = "You are a creative writer.";

    SetLength(oContents, 1);
    oContents[0] = new TsgcGeminiClass_Request_Content();
    oContents[0]->Role = "user";
    SetLength(oParts, 1);
    oParts[0] = new TsgcGeminiClass_Request_Part();
    oParts[0]->Text = "Write a haiku about programming.";
    oContents[0]->Parts = oParts;
    oRequest->Contents = oContents;

    TsgcGeminiClass_Response_GenerateContent *oResponse = Gemini->CreateContent(oRequest);
    try {
        if (Length(oResponse->Candidates) > 0)
```

```
        if (Length(oResponse->Candidates[0]->Parts) > 0)
            ShowMessage(oResponse->Candidates[0]->Parts[0]->Text);
    } __finally {
        delete oResponse;
    }
} __finally {
    delete oParts[0];
    delete oContents[0];
    delete oRequest;
}
```

Gemini | 视觉

Gemini 能够理解和分析图像。您可以在内容部分中以 base64 编码数据的形式发送图像。

视觉示例

将 Base64 编码的图像发送给 Gemini 进行分析。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

// Load image and encode to base64
TFileStream *oStream = new TFileStream("photo.jpg", fmOpenRead);
try {
    TBytesStream *oBytes = new TBytesStream();
    try {
        oBytes->CopyFrom(oStream, 0);
        vImageBase64 = EncodeBase64(oBytes->Memory, oBytes->Size);
    } __finally {
        oBytes->Free();
    }
} __finally {
    oStream->Free();
}

ShowMessage(Gemini->_CreateVisionContent("gemini-2.0-flash",
    "Describe this image", vImageBase64, "image/jpeg"));
```


Gemini | 模型

列出并检索可用 Gemini 模型的相关信息。

列出模型

列出所有可用的模型。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_GetModels());
```

获取模型

获取特定模型的详细信息。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_GetModel("gemini-2.0-flash"));
```

高级示例

使用类型化响应类对模型数据进行完全控制。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

TsgcGeminiClass_Response_Models *oModels = Gemini->GetModels();
try {
    for (int i = 0; i < oModels->Models.Length; i++)
        ShowMessage(oModels->Models[i]->DisplayName + " - " + oModels->Models[i]->Description);
} __finally {
    delete oModels;
}
```

Gemini | 结构化输出

Gemini 可以按照您定义的 Schema 以结构化 JSON 格式返回响应，从而保证输出的 JSON 有效、可解析且符合您的 Schema 定义。

简单示例

使用便捷方法创建带有 JSON schema 输出的内容。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

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

ShowMessage(Gemini->_CreateContentJSON("gemini-2.0-flash",
    "Extract the name and age: John is 30 years old.", vSchema));
```

高级示例

使用类型化类配置带有 ResponseMimeType 和 ResponseSchema 的结构化输出。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

TsgcGeminiClass_Request_GenerateContent *oRequest = new TsgcGeminiClass_Request_GenerateContent();
try {
    oRequest->Model = "gemini-2.0-flash";
    oRequest->MaxOutputTokens = 4096;
    oRequest->ResponseMimeType = "application/json";
    oRequest->ResponseSchema =
        "{\"type\":\"object\",\"properties\":{\"sentiment\":{\"type\":\"string\"}, \"confidence\":{\"type\":\"number\"}}, \"required\": [\"sentiment\", \"confidence\"]}";

    SetLength(oContents, 1);
    oContents[0] = new TsgcGeminiClass_Request_Content();
    oContents[0]->Role = "user";
    SetLength(oParts, 1);
    oParts[0] = new TsgcGeminiClass_Request_Part();
    oParts[0]->Text = "Analyze the sentiment: I love this product!";
    oContents[0]->Parts = oParts;
    oRequest->Contents = oContents;

    TsgcGeminiClass_Response_GenerateContent *oResponse = Gemini->CreateContent(oRequest);
    try {
        if (Length(oResponse->Candidates) > 0)
            if (Length(oResponse->Candidates[0]->Parts) > 0)
                ShowMessage(oResponse->Candidates[0]->Parts[0]->Text);
    } __finally {
        delete oResponse;
    }
} __finally {
    delete oParts[0];
    delete oContents[0];
    delete oRequest;
}
```

属性

- **ResponseMimeType**：设置为 'application/json' 以后用结构化 JSON 输出。
- **ResponseSchema**：定义预期输出结构的 JSON Schema 字符串。

Gemini | 令牌计数

在将消息发送至 API 之前统计其令牌数量。这有助于估算成本并确保消息不超出模型限制。

简单示例

统计消息的令牌数。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->_CountTokens("gemini-2.0-flash", "Hello, how are you?"));
```

高级示例

使用类型化的请求/响应类可完全控制令牌计数参数。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

TsgcGeminiClass_Request_CountTokens *oRequest = new TsgcGeminiClass_Request_CountTokens();
try {
    oRequest->Model = "gemini-2.0-flash";
    SetLength(oContents, 1);
    oContents[0] = new TsgcGeminiClass_Request_Content();
    oContents[0]->Role = "user";
    SetLength(oParts, 1);
    oParts[0] = new TsgcGeminiClass_Request_Part();
    oParts[0]->Text = "Explain quantum computing in simple terms.";
    oContents[0]->Parts = oParts;
    oRequest->Contents = oContents;

    TsgcGeminiClass_Response_CountTokens *oResponse = Gemini->CountTokens(oRequest);
    try {
        ShowMessage("Total tokens: " + IntToStr(oResponse->TotalTokens));
    } __finally {
        delete oResponse;
    }
} __finally {
    delete oParts[0];
    delete oContents[0];
    delete oRequest;
}
```

Gemini | 嵌入

为文本内容生成向量嵌入。嵌入捕获文本的语义含义，可用于相似性搜索、聚类和其他 NLP 任务。

简单示例

为文本生成嵌入向量。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";
ShowMessage(Gemini->EmbedContent("text-embedding-004", "Hello world"));
```

高级示例

使用类型化响应类可完全控制嵌入数据。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

TsgcGeminiClass_Response_Embedding *oEmbedding = Gemini->EmbedContent("text-embedding-004", "Hello world");
try {
    ShowMessage("Dimensions: " + IntToStr(oEmbedding->Values.Length));
    for (int i = 0; i < oEmbedding->Values.Length; i++)
        ShowMessage(FloatToStr(oEmbedding->Values[i]));
} __finally {
    delete oEmbedding;
}
```

Gemini | 工具使用

Gemini 支持函数调用（工具使用），允许您定义模型在对话中可以调用的函数。当 Gemini 决定调用一个函数时，它会在响应中返回一个 `functionCall` 部分。然后您执行该函数并将结果发送回去。

函数调用流程

1. 使用名称、描述和参数（JSON Schema）定义函数声明。
2. 发送包含函数声明的内容生成请求。
3. Gemini 以包含函数名称和参数的 **functionCall** 部分作为响应。
4. 使用提供的参数执行函数。
5. 发送包含 **functionResponse** 部分的新请求，其中包含结果。
6. Gemini 以最终答案作出响应。

示例

定义天气函数并处理函数调用。

```
TsgcHTTP_API_Gemini *Gemini = new TsgcHTTP_API_Gemini(NULL);
Gemini->GeminiOptions->ApiKey = "API_KEY";

// Step 1: Create request with function declarations
TsgcGeminiClass_Request_GenerateContent *oRequest = new TsgcGeminiClass_Request_GenerateContent();
try {
    oRequest->Model = "gemini-2.0-flash";
    oRequest->MaxOutputTokens = 4096;

    // Define user message
    SetLength(oContents, 1);
    oContents[0] = new TsgcGeminiClass_Request_Content();
    oContents[0]->Role = "user";
    SetLength(oParts, 1);
    oParts[0] = new TsgcGeminiClass_Request_Part();
    oParts[0]->Text = "What is the weather in San Francisco?";
    oContents[0]->Parts = oParts;
    oRequest->Contents = oContents;

    // Define function
    SetLength(oFunctions, 1);
    oFunctions[0] = new TsgcGeminiClass_Request_FunctionDeclaration();
    oFunctions[0]->Name = "get_weather";
    oFunctions[0]->Description = "Get the current weather in a given location";
    oFunctions[0]->Parameters =
        "{\n\"type\": \"object\", \"properties\": {\n\"location\": {\n\"type\": \"string\", \"\n\"description\": \"The city and state\"}, \"required\": [\"location\"]}";
    oRequest->FunctionDeclarations = oFunctions;

    // Step 2: Send request
    TsgcGeminiClass_Response_GenerateContent *oResponse = Gemini->CreateContent(oRequest);
    try {
        // Step 3: Check for function call
        if (Length(oResponse->Candidates) > 0) {
            if (Length(oResponse->Candidates[0]->Parts) > 0) {
                if (oResponse->Candidates[0]->Parts[0]->FunctionCallName != "") {
                    vFunctionName = oResponse->Candidates[0]->Parts[0]->FunctionCallName;
                    vFunctionArgs = oResponse->Candidates[0]->Parts[0]->FunctionCallArgs;
                    // Step 4: Execute function and get result
                    vResult = "72 degrees and sunny";
                }
            }
        }
    } finally {
        delete oResponse;
    }
} finally {
    delete oParts[0];
    delete oContents[0];
    delete oFunctions[0];
    delete oRequest;
}
```

属性

- **FunctionDeclarations**：定义可用函数的 TsgcGeminiClass_Request_FunctionDeclaration 数组。
 - **Name**：函数名称。
 - **Description**：描述该函数的功能。
 - **Parameters**：定义函数参数的 JSON Schema 字符串。
- **ToolChoice**：控制函数调用行为。设置模式（例如"AUTO"、"ANY"、"NONE"）。

Ollama

Ollama 是一款用于在本地运行大型语言模型的开源工具。它支持包括 Llama、Mistral、Gemma、Phi 等在内的多种模型，无需云 API 访问即可实现本地 AI 推理。

sgcWebSockets 库提供 Delphi 组件 **TsgcHTTP_API_Ollama** 用于与 Ollama API 交互。

Ollama API

Ollama API 提供对本地运行模型的访问，用于 AI 驱动的应用程序。该 API 支持文本生成、流式传输、模型管理和嵌入向量。默认情况下无需 API 密钥，因为模型在本地运行。主机和端口均可配置。

功能特性

- 消息
 - [Ollama 消息示例](#)
- 模型
 - [Ollama 模型示例](#)
- 嵌入
 - [Ollama 嵌入示例](#)

配置

Ollama 在本地运行，默认情况下监听 **http://localhost:11434**。在 **OllamaOptions.Host** 属性中配置主机。API 密钥是可选的，仅在您的 Ollama 实例配置了身份验证时才需要。

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
```

消息

发送带有文本内容的结构化输入消息列表，模型将生成对话中的下一条消息。

- **_CreateMessage**：使用指定的模型和用户提示创建消息。
 - **Model**：要使用的模型（例如 llama3）。
 - **Message**：用户消息内容。
- **_CreateMessageWithSystem**：创建带有系统提示词的消息。
 - **System**：设置助手行为的系统提示词。
- **_CreateMessageStream**：创建启用流式传输（SSE）的消息。事件通过 OnHTTPAPISSE 事件处理程序传递。

模型

管理和查询本地可用的模型。

- **_GetModels**：列出所有可用模型。
- **_GetTags**：列出所有本地可用的模型标签。
- **_ShowModel**：检索特定模型的详细信息。
 - **Model**：要查询的模型名称。
- **_PullModel**：从 Ollama 模型库下载模型。
 - **Model**：要拉取的模型名称。
- **_DeleteModel**：删除本地可用的模型。
 - **Model**：要删除的模型名称。

嵌入向量

使用本地运行的模型从文本输入生成向量嵌入。

- **_CreateEmbeddings**：为给定文本生成嵌入向量。
 - **Model**：用于生成嵌入向量的模型（例如 llama3）。

- **Input**：要生成嵌入向量的文本。

TsgcHTTP_API_Ollama

用于本地聊天补全和嵌入向量的 Ollama REST API 客户端。

简介

用于本地聊天补全和嵌入向量的 Ollama REST API 客户端。

简介

用于本地聊天补全和嵌入向量的 Ollama REST API 客户端。
使用您的 API 密钥及可选的 HTTP/重试/日志设置配置 **OllamaOptions** 属性，然后调用类型化方法（如 `CreateMessage/CreateChatCompletion`），传入请求对象并接收类型化响应类。
有关该提供商的背景信息，请参阅 [Ollama](#) 指南。

参考

- 属性 — 组件上提供 5 个属性。
- 方法 — 2 个公共方法。
- 事件 — 2 个可处理的事件。

Ollama | 消息

发送带有文本内容的结构化输入消息列表，模型将生成对话中的下一条消息。

简单示例

向本地 Ollama 模型发送 Hello 消息。

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_CreateMessage("llama3", "Hello!"));
```

系统提示词示例

发送带有系统提示的消息以控制模型行为。

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_CreateMessageWithSystem("llama3",
    "You are a helpful assistant that responds in Spanish.",
    "What is the capital of France?"));
```

流式示例

使用 Server-Sent Events (SSE) 实时流式传输响应。分配 OnHTTPAPISSE 事件处理程序以接收流式事件。

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
Ollama->OnHTTPAPISSE = OnSSEEvent;
Ollama->_CreateMessageStream("llama3", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
    const String aData, bool &Cancel)
{
    // aEvent contains the event type
    // aData contains the JSON data for this event
    Memo1->Lines->Add(aData);
}
```

Ollama | 模型

管理和查询本地可用的 Ollama 模型。

列出模型

列出所有可用的模型。

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_GetModels());
```

获取标签

列出所有本地可用的模型标签。

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_GetTags());
```

显示模型

检索特定模型的详细信息。

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_ShowModel("llama3"));
```

拉取模型

从 Ollama 模型库下载模型。

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_PullModel("llama3"));
```

Ollama | 嵌入

使用本地运行的 Ollama 模型从文本输入生成向量嵌入。嵌入可用于语义搜索、聚类和其他自然语言处理任务。

创建嵌入向量

为给定的文本输入生成嵌入。

```
TsgcHTTP_API_Ollama *Ollama = new TsgcHTTP_API_Ollama(NULL);
Ollama->OllamaOptions->Host = "http://localhost:11434";
ShowMessage(Ollama->_CreateEmbeddings("llama3", "Hello world"));
```

Grok

Grok 是由 xAI 开发的对话式 AI 助手，旨在提供有帮助且信息丰富的响应，并具有实时知识访问能力。Grok 模型以其强大的推理能力和最新的信息访问而著称。

sgcWebSockets 库提供了一个 Delphi 组件 **TsgcHTTP_API_Grok**，用于与 Grok API 进行交互。

Grok API

Grok API 提供对 Grok 模型的访问，用于构建 AI 驱动的应用程序。该 API 支持文本生成、视觉（图像理解）、流式传输和模型列表。

功能特性

- 消息
 - [Grok 消息示例](#)
- 视觉
 - [Grok Vision 示例](#)
- 模型
 - [Grok 模型示例](#)

配置

Grok API 使用 API 密钥进行身份验证。访问您的 [xAI 控制台](#) 获取请求中使用的 API 密钥。

请记住，您的 API 密钥是机密！请勿与他人共享或在任何客户端代码中公开它。

此 **API Key** 必须在组件的 **GrokOptions.ApiKey** 属性中配置。

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "YOUR_API_KEY";
```

消息

发送带有文本内容的结构化输入消息列表，模型将生成对话中的下一条消息。

- **_CreateMessage**：使用指定模型和用户提示词创建消息。
 - **Model**：要使用的模型（例如 grok-3）。
 - **Message**：用户消息内容。
- **_CreateMessageWithSystem**：创建带有系统提示词的消息。
 - **System**：设置助手行为的系统提示词。
- **_CreateMessageStream**：创建启用流式传输（SSE）的消息。事件通过 OnHTTPAPISSE 事件处理程序传递。

视觉

Grok 可以理解以 base64 编码内容形式传递在消息中的图像。

- **_CreateVisionMessage**：发送带有文本提示的图像。
 - **ImageBase64**：Base64 编码的图像数据。
 - **MediaType**：MIME 类型（image/jpeg、image/png、image/gif、image/webp）。
 - **Prompt**：随图像发送的文本提示。

模型

列出可用的 Grok 模型。

- **_GetModels**：列出所有可用模型。

TsgcHTTP_API_Grok

TsgcHTTP_API_Grok 组件。

简介

用于聊天补全和模型列表的 xAI Grok REST API 客户端。

简介

用于聊天补全和模型列表的 xAI Grok REST API 客户端。

使用您的 API 密钥和可选的 HTTP/重试/日志设置配置 **GrokOptions** 属性，然后使用请求对象调用 `CreateMessage/`
`CreateChatCompletion` 等类型化方法，并接收类型化的响应类。

有关该提供商的背景信息，请参阅 [Grok](#) 指南。

参考

- 属性 — 该组件提供 7 个属性。
- 方法 — 1 个公共方法。
- 事件 — 0 个可处理的事件。

Grok | 消息

发送带有文本内容的结构化输入消息列表，模型将生成对话中的下一条消息。

简单示例

向 Grok 发送 Hello 消息。

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "API_KEY";
ShowMessage(Grok->_CreateMessage("grok-3", "Hello!"));
```

系统提示词示例

发送带有系统提示的消息以控制 Grok 的行为。

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "API_KEY";
ShowMessage(Grok->_CreateMessageWithSystem("grok-3",
    "You are a helpful assistant that responds in Spanish.",
    "What is the capital of France?"));
```

流式示例

使用 Server-Sent Events (SSE) 实时流式传输响应。分配 OnHTTAPISSE 事件处理程序以接收流式事件。

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "API_KEY";
Grok->OnHTTAPISSE = OnSSEEvent;
Grok->_CreateMessageStream("grok-3", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
    const String aData, bool &Cancel)
{
    Memo1->Lines->Add(aData);
}
```

Grok | 视觉

Grok 可以理解在消息中以 Base64 编码内容传递的图像。使用视觉端点发送图像以及文本提示并接收描述或分析。

视觉示例

向 Grok 发送 base64 编码的图像以进行分析。

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "API_KEY";
ShowMessage(Grok->_CreateVisionMessage("grok-2-vision",
    "Describe this image.", vBase64, "image/jpeg"));
```


Grok | Models

列出可用的 Grok 模型及其基本信息。

列出模型示例

检索所有可用 Grok 模型的列表。

```
TsgcHTTP_API_Grok *Grok = new TsgcHTTP_API_Grok(NULL);
Grok->GrokOptions->ApiKey = "API_KEY";
ShowMessage(Grok->_GetModels);
```

Mistral AI

Mistral AI 是一家法国人工智能公司，专注于开发高效且强大的语言模型。其模型以在推理、编程和多语言任务上的出色表现而闻名，同时保持有竞争力的效率。

sgcWebSockets 库提供 Delphi 组件 **TsgcHTTP_API_Mistral**，用于与 Mistral AI API 交互。

Mistral API

Mistral API 提供对 Mistral 模型的访问，用于构建 AI 驱动的应用程序。该 API 支持文本生成、视觉（图像理解）、流式传输、嵌入、JSON 模式和模型列表。

功能特性

- 消息
 - [Mistral 消息示例](#)
- Vision
 - [Mistral Vision 示例](#)
- 模型
 - [Mistral 模型示例](#)
- 嵌入
 - [Mistral 嵌入示例](#)

配置

Mistral API 使用 API 密钥进行身份验证。请访问 Mistral Console 中的 [API 密钥](#) 页面，获取您在请求中使用的 API 密钥。

请记住，您的 API 密钥是机密！请勿与他人共享或在任何客户端代码中公开它。

此 **API Key** 必须在组件的 **MistralOptions.ApiKey** 属性中配置。

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "YOUR_API_KEY";
```

消息

发送带有文本内容的结构化输入消息列表，模型将生成对话中的下一条消息。

- **_CreateMessage**：使用指定的模型和用户提示创建消息。
 - **Model**：要使用的模型（例如 mistral-large-latest）。
 - **Message**：用户消息内容。
- **_CreateMessageWithSystem**：创建带有系统提示词的消息。
 - **System**：设置助手行为的系统提示词。
- **_CreateMessageStream**：创建启用流式传输（SSE）的消息。事件通过 OnHTTPAPISSE 事件处理程序传递。
- **_CreateMessageJSON**：在启用 JSON 模式的情况下创建消息。模型将返回有效的 JSON 响应。
 - **Model**：要使用的模型。
 - **Message**：用户消息内容。

视觉

Mistral 能够理解消息中以 base64 编码内容形式传入的图像。

- **_CreateVisionMessage**：发送带有文本提示的图像。
 - **ImageBase64**：Base64 编码的图像数据。
 - **MediaType**：MIME 类型（image/jpeg、image/png、image/gif、image/webp）。
 - **Prompt**：随图像发送的文本提示。

模型

列出可用的 Mistral 模型。

- **_GetModels**：列出所有可用模型。

嵌入向量

获取给定输入的向量表示，可用于语义搜索、聚类及其他机器学习任务。

- **_CreateEmbeddings**：创建表示输入文本的嵌入向量。
 - **Model**：要使用的模型（例如 mistral-embed）。
 - **Input**：需要获取嵌入向量的输入文本。

TsgcHTTP_API_Mistral

Mistral REST API 客户端，用于聊天完成和嵌入。

简介

Mistral REST API 客户端，用于聊天完成和嵌入。

简介

Mistral REST API 客户端，用于聊天完成和嵌入。
使用您的 API 密钥和可选的 HTTP/重试/日志设置配置 **MistralOptions** 属性，然后调用类型化方法（例如 `CreateMessage/CreateChatCompletion`），传入请求对象并接收类型化的响应类。
有关该提供商的背景信息，请参阅 [Mistral](#) 指南。

参考

- 属性 — 组件上提供 5 个属性。
- 方法 — 2 个公共方法。
- 事件 — 2 个可处理的事件。

Mistral | 消息

发送带有文本内容的结构化输入消息列表，模型将生成对话中的下一条消息。

简单示例

向 Mistral 发送一条 Hello 消息。

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_CreateMessage("mistral-large-latest", "Hello!"));
```

系统提示词示例

发送带有系统提示的消息以控制 Mistral 的行为。

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_CreateMessageWithSystem("mistral-large-latest",
    "You are a helpful assistant that responds in Spanish.",
    "What is the capital of France?"));
```

流式示例

使用 Server-Sent Events (SSE) 实时流式传输响应。分配 OnHTTPAPISSE 事件处理程序以接收流式事件。

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
Mistral->OnHTTPAPISSE = OnSSEEvent;
Mistral->_CreateMessageStream("mistral-large-latest", "Tell me a story.");

void __fastcall TForm1::OnSSEEvent(TObject *Sender, const String aEvent,
    const String aData, bool &Cancel)
{
    Memo1->Lines->Add(aData);
}
```

JSON 模式示例

请求 Mistral 返回 JSON 格式的响应。

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_CreateMessageJSON("mistral-large-latest",
    "List 3 colors as JSON"));
```

Mistral | 视觉

Mistral 能够理解消息中以 base64 编码内容形式传入的图像。使用视觉端点发送图像以及文本提示，并接收描述或分析。

视觉示例

向 Mistral 发送 base64 编码的图像进行分析。

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_CreateVisionMessage("mistral-large-latest",
    "Describe this image.", vBase64, "image/jpeg"));
```

Mistral | 模型

列出可用的 Mistral 模型及其基本信息。

列出模型示例

检索所有可用 Mistral 模型的列表。

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);  
Mistral->MistralOptions->ApiKey = "API_KEY";  
ShowMessage(Mistral->_GetModels);
```

Mistral | 嵌入

获取给定输入的向量表示，可用于语义搜索、聚类及其他机器学习任务。

嵌入向量示例

为文本输入创建嵌入向量。

```
TsgcHTTP_API_Mistral *Mistral = new TsgcHTTP_API_Mistral(NULL);
Mistral->MistralOptions->ApiKey = "API_KEY";
ShowMessage(Mistral->_CreateEmbeddings("mistral-embed", "Hello world"));
```


IoT

物联网（IoT）是指将互联网连接从个人计算机和移动设备等传统计算平台，延伸至各类传统“哑”设备或非联网物理设备和日常物品的概念。这些嵌入了电子设备、互联网连接及其他硬件（如传感器）的设备，可通过互联网与其他设备通信和交互，并可被远程监控和控制。

sgcWebSockets 包实现了以下 IoT 客户端：

1. Amazon AWS IoT：AWS IoT 在联网设备（如传感器、执行器、嵌入式微控制器或智能家电）与 AWS 云之间提供安全的双向通信。这使您能够从多个设备收集遥测数据，并对数据进行存储和分析。您还可以创建允许用户通过手机或平板电脑控制这些设备的应用程序。

2. Azure IoT Hub：IoT Hub 是托管在云中的托管服务，充当 IoT 应用程序与其管理的设备之间双向通信的中央消息枢纽。您可以使用 Azure IoT Hub 构建 IoT 解决方案，在数百万台 IoT 设备与云托管的解决方案后端之间实现可靠、安全的通信。

IoT Amazon MQTT 客户端

什么是 AWS IoT ?

AWS IoT 在联网设备（如传感器、执行器、嵌入式微控制器或智能设备）与 AWS 云之间提供安全的双向通信。这使您可以从多台设备收集遥测数据并存储和分析。您还可以创建应用程序，使用户能够通过手机或平板电脑控制这些设备。

消息代理

提供一种安全机制，使设备和 AWS IoT 应用程序能够相互发布和接收消息。您可以直接使用 MQTT 协议或通过 WebSocket 上的 MQTT 进行发布和订阅。

AWS IoT 消息代理是一个发布/订阅代理服务，能够向 AWS IoT 发送和接收消息。与 AWS IoT 通信时，客户端将消息发送到类似 `Sensor/temp/room1` 这样的主题地址。

消息代理随后将消息发送给所有已注册接收该主题消息的客户端。发送消息的行为称为发布，注册接收特定主题过滤器消息的行为称为订阅。

主题命名空间针对每个 AWS 账户和区域是隔离的。例如，某个 AWS 账户的 `Sensor/temp/room1` 主题与另一个 AWS 账户的同名主题相互独立。区域也是如此。同一 AWS 账户在 `us-east-1` 的 `Sensor/temp/room1` 主题与 `us-east-2` 的同名主题相互独立。AWS IoT 不支持跨 AWS 账户和区域发送及接收消息。

消息代理维护所有客户端会话及每个会话的订阅列表。当消息发布到某个主题时，代理检查具有映射到该主题的订阅的会话，然后将发布消息转发给所有当前有已连接客户端的会话。

MQTT 客户端

TsgcloTAmazon_MQTT_Client 是用于连接 AWS IoT 的组件。一个客户端只能连接到一个设备。客户端使用普通 MQTT 协议连接，并使用 X.509 客户端证书进行身份验证。

连接到 AWS IoT 时，客户端需要以下属性：

Amazon.ClientId：客户端标识符，可选。

Amazon.Endpoint：MQTT 客户端将连接的服务器名称。

Amazon.Port：默认使用端口 8883。如果端口为 443，则自动使用 ALPN 进行连接（需要自定义 Indy 版本）。

AWS IoT Core 支持使用 MQTT 和基于 WebSocket Secure (WSS) 的 MQTT 协议的设备和客户端进行消息发布和订阅。下表列出了 AWS IoT 设备端点支持的协议以及它们使用的身份验证方法和端口。

协议	身份验证	端口	ALPN 协议名称
WebSocket 上的 MQTT	Signature Version 4	443	
WebSocket 上的 MQTT	自定义身份验证	443	
MQTT	X.509 客户端证书	443	x-amzn-mqtt-ca
MQTT	X.509 客户端证书	8883	
MQTT	自定义身份验证	443	mqtt

证书身份验证

您需要在 Amazon AWS 控制台中创建证书，并设置其存储路径。

使用 **OpenSSL** 作为 IOHandler 时，必须在以下路径中设置证书

Certificate.Enabled：如果要使用证书，请设置为 True。
Certificate.CertFile：X.509 客户端证书的路径。
Certificate.KeyFile：X.509 客户端密钥文件的路径。

使用 **SChannel** 作为 IOHandler 时，请先将 PEM 证书 + 私钥转换为 PFX 证书。这需要 OpenSSL 二进制文件：

```
openssl pkcs12 -inkey 884ccf73ff-private.pem.key -in 884ccf73ff-certificate.pem.crt -export -out 884ccf73ff-certi
```

然后设置以下路径（无需设置密钥文件，因为它已包含在证书中）。

Certificate.Enabled：如果要使用证书，请设置为 True。
Certificate.CertFile：PFX 证书的路径

SignatureV4 身份验证

您需要在 Amazon AWS 控制台中创建一个用户，并保存 Access 和 Secret 密钥，这些密钥将用于签署 WebSocket 请求。

SignatureV4.Enabled：若要使用此类型的身份验证，请将其设为 True。
SignatureV4.Region：您的设备所在的区域（例如：us-east-1）。
SignatureV4.AccessKey：在 Amazon 控制台中创建的访问密钥或作为临时凭证获取的访问密钥。
SignatureV4.SecretKey：在 Amazon 控制台中创建的密钥，或作为临时凭证获取
SignatureV4.SessionToken：（条件性）如果您正在使用临时安全凭证，请在此处设置安全令牌。
OpenSSL_Options：OpenSSL 库的配置。
APIVersion：允许定义将使用哪个 OpenSSL API 版本。
oslAPI_1_0：使用 API 1.0 OpenSSL，这是 Indy 支持的最新版本
oslAPI_1_1：使用 OpenSSL API 1.1，需要我们定制的 Indy 库，并支持 OpenSSL 1.1.1 库（含 TLS 1.3 支持）。
oslAPI_3_0：使用 API 3.0 OpenSSL，需要我们的自定义 Indy 库，并允许使用 OpenSSL 3.0.0 库（支持 TLS 1.3）。
LibPath：在此处您可以配置 openssl 库的位置
oslpNone：这是默认值，OpenSSL 库应与二进制文件位于同一文件夹中，或位于已知路径中。
oslpDefaultFolder：自动设置所有 IDE 版本应查找 OpenSSL 库的路径。
oslpCustomFolder：如果选择此选项，请在属性 LibPathCustom 中定义完整路径。
LibPathCustom：当 LibPath = oslpCustomFolder 时，在此处定义 openssl 库所在的完整路径。
UnixSymLinks：启用或禁用在 Unix 系统下加载符号链接（默认启用，OSX64 除外）：
oslsSymLinksDefault：默认情况下启用，但在 OSX64 下除外（MacOS Monterey 之后，不带版本号尝试加载库会失败）。
oslsSymLinksLoadFirst：先加载符号链接，然后再尝试加载版本库。
oslsSymLinksLoad：在尝试加载版本库后加载符号链接。
oslsSymLinksDontLoad：不加载符号链接。

*SignatureV4 需要 Indy 10.5.7 及以上版本。

自定义身份验证

自定义认证使您能够通过授权器资源定义客户端的身份验证和授权方式。设备可通过请求头字段或查询参数（用于基于 WebSockets 的 MQTT 协议）传递凭证，或在 MQTT CONNECT 消息的用户名和密码字段中传递（用于 MQTT 及基于 WebSockets 的 MQTT 协议）。

CustomAuthentication.Enabled：若要使用此类身份验证，请设置为 True。

CustomAuthentication.Parameters：在此设置将传递给服务器的查询参数（默认为 /mqtt）

CustomAuthentication.Headers：在此处可以设置自定义头字段。

CustomAuthentication.WebSockets：若设置为 true，连接将通过 WebSocket 协议工作，否则将通过纯 TCP 工作。

MQTTAuthentication.Enabled：如果您需要在 MQTT 连接中传递用户名/密码，请启用此属性。

MQTTAuthentication.Username：MQTT 连接的用户名

MQTTAuthentication.Password：MQTT 连接的密钥。

客户端可以选择发送 ClientId 以标识客户端连接，其他客户端可以订阅以在每次该客户端连接、订阅、断开连接时收到通知。

授权

如果无法使用 8883 端口并使用 TCP 作为传输协议（这是默认设置），Amazon 将通过"AWS IoT Core 策略"向客户端和订阅提供或拒绝授权。您可能需要授权您的客户端 id。

登录您的 Amazon AWS 控制台，进入 IoT Core，访问菜单 "Secure/Policies"，选择附加到您 IoT Thing 的策略，并在末尾查看连接的配置方式。示例：

```
{
  "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-*"
  ]
}
```

此配置表示只允许 ID 为 sdk-java、basicPubSub 和 sdk-nodejs-* 的客户端连接。请相应地更改并重试。如果仍然无法连接，请启用日志并在 CloudWatch 中检查无法连接的原因。

其他属性

MQTTHeartBeat：若启用，尝试通过每 x 秒发送一次 ping 来保持 MQTT 连接活跃。

Interval：每次 ping 之间的秒数。

MQTTAuthentication：如果启用，将在 MQTT 连接中包含用户名和密码

UserName：用户名称

密码：密钥字符串

WatchDog：如果启用，当检测到意外断开连接时，将尝试自动重新连接到服务器。

Interval：两次重连尝试之间的秒数。

Attempts：最大重连尝试次数；零表示无限制。

LogFile：若启用，将套接字消息保存到日志文件（对调试有用）。如果从多个线程访问，日志文件的访问不是线程安全的。

Enabled：如果启用，每次套接字接收和发送消息时，都会将其保存到文件中。

FileName：文件的完整路径。

实现

Amazon MQTT 实现基于 MQTT 3.1.1 版本，但在以下方面偏离了规范：

- 在 AWS IoT 中，以服务质量 (QoS) 0 订阅主题意味着消息将被传递零次或多次。消息可能被传递多次，且多次传递时可能使用不同的数据包 ID。在这种情况下，不设置 DUP 标志。
- AWS IoT 不支持使用 QoS 2 进行发布和订阅。当请求 QoS 2 时，AWS IoT 消息代理不会发送 PUBACK 或 SUBACK。
- 响应连接请求时，消息代理发送一条 CONNACK 消息。此消息包含一个标志，指示连接是否在恢复之前的会话。如果两个 MQTT 客户端同时使用相同的客户端 ID 连接，此标志的值可能不正确。
- 当客户端订阅某个主题时，消息代理发送 SUBACK 与客户端开始接收新的匹配消息之间可能存在延迟。
- MQTT 规范为发布者提供了一项功能，允许请求 broker 保留发送到主题的最后一条消息，并将其发送给所有未来的主题订阅者。AWS IoT 不支持保留消息。如果发出保留消息的请求，连接将被断开。
- 消息代理使用客户端 ID 来标识每个客户端。客户端 ID 作为 MQTT 负载的一部分从客户端传递到消息代理。不允许具有相同客户端 ID 的两个客户端同时连接到消息代理。当客户端使用另一个客户端正在使用的客户端 ID 连接到消息代理时，将向两个客户端发送 CONNACK 消息，当前已连接的客户端将被断开连接。
- 在极少数情况下，消息代理可能会以不同的数据包 ID 重新发送相同的逻辑 PUBLISH 消息。
- 消息代理不保证消息和 ACK 的接收顺序。

连接到 AWS IoT

首先，您需要登录 AWS 控制台，注册一个新设备并为该设备创建 X.509 证书。完成后，您可以创建一个新的 TsgcIoTAmazon_MQTT_Client 并连接到 AWS IoT 服务器。例如：

```
oClient = new TsgcIoTAmazon_MQTT_Client();
oClient->Amazon->Endpoint = "a2ohgdjqitsmij-ats.iot.us-west-2.amazonaws.com";
oClient->Amazon->ClientId = "sgcWebSockets";
oClient->Certificate->CertFile = "amazon-certificate.pem.crt";
oClient->Certificate->KeyFile = "amazon-private.pem.key";
oClient->OnMQTTConnect = OnMQTTConnectEvent;
oClient->Active = true;

void OnMQTTConnect(TsgcWSConnection *Connection, const bool Session, const TmqttConnReturnCode
ReturnCode)
{
    ShowMessage("Connected to AWS");
}
```

主题

消息 broker 使用主题将消息从发布客户端路由到订阅客户端。正斜杠 (/) 用于分隔主题层次结构。下表列出了订阅时可在主题过滤器中使用的通配符。# 必须是订阅主题中的最后一个字符。它作为通配符匹配当前树和所有子树。

例如，订阅 Sensor/# 将接收发布至 Sensor/、Sensor/temp、Sensor/temp/room1 的消息，但不接收发布至 Sensor 的消息。

+ 精确匹配主题层级中的一个元素。例如，订阅 Sensor/+room1 将接收发布到 Sensor/temp/room1、Sensor/moisture/room1 等主题的消息。

```
oClient = new TsgcIoTAmazon_MQTT_Client();
...
oClient->OnSubscribe = OnSubscribeEvent;

vPacketIdentifier = oClient->Subscribe("Sensor/moisture/room1");

void OnMQTTSubscribe(TsgcWSConnection *Connection, Word aPacketIdentifier, TsgcWSSUBACKS *aCodes)
{
    if (vPacketIdentifier == aPacketIdentifier)
    {
        ShowMessage("Subscribed to topic Sensor/moisture/room1");
    }
}

// Client, can send a message using Publish method.
oClient->Publish("Sensor/moisture/room1", "{\"temp\"=10}");

// Messages received from server, are dispatched OnMQTTPublishEvent.
```



```
// For extended payload access (string, bytes or stream), use OnMQTTPublishEx.
void OnMQTTPublish(TsgcWSConnection *Connection, string aTopic, string aText)
{
    DoLog("Received Message: " + aTopic + " " + aText);
}
```

保留主题

以下方法用于订阅/发布到保留主题。

Subscribe_ClientConnected(const aClientId: String)：当具有指定客户端 ID 的 MQTT 客户端连接到 AWS IoT 时，AWS IoT 会向此主题发布内容

Subscribe_ClientDisconnected(const aClientId: String)：当具有指定客户端 ID 的 MQTT 客户端断开与 AWS IoT 的连接时，AWS IoT 将发布到此主题。

Subscribe_ClientSubscribed(const aClientId: String)：当具有指定客户端 ID 的 MQTT 客户端订阅 MQTT 主题时，AWS IoT 将发布到此主题。

Subscribe_ClientUnsubscribed(const aClientId: String)：当具有指定客户端 ID 的 MQTT 客户端取消订阅某个 MQTT 主题时，AWS IoT 会向此主题发布消息。

Publish_Rule(const aRuleName, aText: String)：设备或应用程序发布到此主题以直接触发规则。

Publish_DeleteShadow(const aThingName, aText: String)：设备或应用程序发布到此主题以删除影子

Subscribe_DeleteShadow(const aThingName: String)：设备或应用程序订阅此主题以删除 Shadow

Subscribe_ShadowDeleted(const aThingName: String)：当影子被删除时，Device Shadow 服务向此主题发送消息

Subscribe_ShadowRejected(const aThingName: String)：当删除影子的请求被拒绝时，Device Shadow 服务向此主题发送消息

Publish_ShadowGet(const aThingName, aText: String)：应用程序或设备向此主题发布空消息以获取影子

Subscribe_ShadowGet(const aThingName: String)：应用程序或设备订阅此主题以获取 shadow

Subscribe_ShadowGetAccepted(const aThingName: String)：当成功发出获取设备影子的请求时，Device Shadow 服务向此主题发送消息

Subscribe_ShadowGetRejected(const aThingName: String)：当对 Shadow 的请求被拒绝时，Device Shadow 服务向此主题发送消息

Publish_ShadowUpdate(const aThingName, aText: String)：设备或应用程序向此主题发布以更新设备影子

Subscribe_ShadowUpdateAccepted(const aThingName: String)：当影子更新成功完成时，Device Shadow 服务会向此主题发送消息

Subscribe_ShadowUpdateRejected(const aThingName: String)：当影子更新被拒绝时，Device Shadow 服务会向此主题发送消息

Subscribe_ShadowUpdateDelta(const aThingName: String)：当在影子的 reported 和 desired 部分之间检测到差异时，Device Shadow 服务向此主题发送消息

Subscribe_ShadowUpdateDocuments(const aThingName: String)：每当成功更新 shadow 时，AWS IoT 将状态文档发布到此主题

持久会话

持久会话表示与 MQTT 消息 broker 的持续连接。当客户端使用持久会话连接到 AWS IoT 消息 broker 时，消息 broker 将保存客户端在连接期间进行的所有订阅。客户端断开连接后，消息 broker 会存储未确认的 QoS 1 消息以及发布到客户端已订阅主题的新 QoS 1 消息。当客户端重新连接到持久会话时，所有订阅将恢复，所有存储的消息将以每秒最多 10 条的速率发送给客户端。

通过在 **OnMQTTBeforeConnect** 事件中将 **cleanSession** 参数设置为 **False**，可创建 MQTT 持久会话。如果客户端没有现有会话，则创建新的持久会话；如果客户端已有会话，则恢复该会话。

设备需要查看 **OnMQTTConnect** 事件中的 **Session** 属性，以确定是否存在持久会话。如果 **Session** 为 **True**，则存在持久会话，并将存储的消息传递给客户端。如果 **Session** 为 **False**，则不存在持久会话，客户端必须重新订阅其主题过滤器。

持久会话的默认过期时间为 1 小时。过期时间从消息代理检测到客户端断开连接（MQTT 断开或超时）时开始计算。持久会话过期时间可通过标准限额增加流程延长。如果客户端在过期时间内未恢复会话，则会话将被终止，所有关联的存储消息将被丢弃。过期时间是近似值，会话可能比配置的持续时间延长最多 30 分钟（但不会提前）。

临时凭证

AWS IoT Core 可以使用通过 Identity Pools 获取的临时凭据，有 2 种类型的身份：

- **UnAuthenticated**：只需要在 IAM 中设置策略类型
- **Authenticated**：需要在 IAM 和 AWS IoT Core 策略中设置策略类型

未认证

如果您使用未认证的凭证，只需在 IAM 菜单中自动创建的未认证角色上附加策略。然后配置客户端，设置由 Cognito 服务返回的 Access、Secret Key 和 Token。

以下是获取未认证凭据的 .NET 代码

```
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;
```

已认证

已认证的凭据，需要在 IAM 菜单中自动创建的已认证角色附加策略，并将用户策略附加到 AWS IoT Core 策略。因此，在 IoT Core 策略菜单中创建新策略，每次有新用户进行身份验证时，将此策略附加到该用户。您可以使用以下 AWS 命令附加策略或创建 lambda 函数。

```
aws iot attach-policy --policy-name PolicyName --target us-east-1:XXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
```

设备预配

Fleet Provisioning 服务支持以下 MQTT API 操作：

- **CreateCertificateFromCsr**：从证书签名请求（CSR）创建证书。AWS IoT 提供由 Amazon 根证书颁发机构（CA）签名的客户端证书。新证书具有 PENDING_ACTIVATION 状态。当您调用 RegisterThing 使用此证书预置设备时，证书状态将变为 ACTIVE 或 INACTIVE，如模板中所述。
- **CreateKeysAndCertificate**：创建新的密钥和证书。AWS IoT 提供由 Amazon 根证书颁发机构（CA）签名的客户端证书。新证书的状态为 PENDING_ACTIVATION。当您调用 RegisterThing 使用此证书预置设备时，证书状态将根据模板变更为 ACTIVE 或 INACTIVE。
- **RegisterThing**：使用预定义模板预置设备。

CreateCertificateFromCsr

使用 CreateCertificateFromCsr 方法，将 CertificateSigningRequest 作为参数传入，以创建证书。为接收此请求的响应，请先订阅以下方法：SubscribeCreateCertificateFromCsrResponse 和 SubscribeCreateCertificateFromCsrError。

CreateKeysAndCertificate

使用 CreateKeysAndCertificate 方法创建新证书和密钥。为了接收此请求的响应，请先订阅以下方法：SubscribeCreateKeysAndCertificateResponse 和 SubscribeCreateKeysAndCertificateError

RegisterThing

使用方法 RegisterThing 注册新设备，将模板名称和 JSON 格式的负载作为参数传入。要接收对此请求的响应，请先订阅以下方法：SubscribeRegisterThingResponse 和 SubscribeRegisterThingError。

IoT Azure MQTT 客户端

什么是 Azure IoT Hub？

IoT Hub 是托管在云中的托管服务，充当 IoT 应用程序与其管理的设备之间双向通信的中央消息枢纽。您可以使用 Azure IoT Hub 构建与数百万 IoT 设备及云托管解决方案后端之间通信可靠且安全的 IoT 解决方案。几乎任何设备都可以连接到 IoT Hub。

IoT Hub 支持从设备到云以及从云到设备的双向通信。IoT Hub 支持多种消息传递模式，例如设备到云的遥测、设备的文件上传以及从云控制设备的请求-响应方法。IoT Hub 监控通过跟踪设备创建、设备故障和设备连接等事件来帮助维护解决方案的健康状态。

IoT Hub 的功能帮助您构建可扩展的全功能 IoT 解决方案，例如管理制造业中使用的工业设备、追踪医疗保健中的宝贵资产以及监控办公楼的使用情况。

消息代理

IoT Hub 为您的设备提供安全的通信通道以发送数据。IoT Hub 和设备 SDK 支持以下用于连接设备的协议：

- **MQTT**
- **MQTT over WebSockets**

多种认证类型支持各种设备能力：

- **SAS** 基于令牌的身份验证，可快速开始使用您的物联网解决方案。
- 单独的 **X.509 证书** 身份验证，用于安全的、基于标准的认证。

MQTT 客户端

TsgcIoT Azure MQTT Client 是用于连接 Azure IoT 的组件。一个客户端只能连接到一个设备。该客户端使用纯 MQTT 协议连接，并通过 SAS / X.509 客户端证书进行身份验证。

为了连接到 Azure IoT Hub，客户端需要以下属性：

Azure.IoTHub：MQTT 客户端将连接的服务器名称。
Azure.Deviceld：Azure IoT Hub 中设备的名称。

Azure 支持多种认证类型，默认使用 SAS 令牌。

SAS 认证

SAS.Enabled：若身份验证使用 SAS，请启用此选项。
SAS.SecretKey：来自您 Azure IoT 账户的 SAS 令牌。
SAS.KeyName：共享访问密钥名称。
SAS.Expiry：设置 SAS 令牌过期前的分钟数。默认值为 1440（24 小时）。

如果您有连接字符串，可以使用 **ReadConnectionString** 方法自动读取连接字符串的值。示例：

```
ReadConnectionString('HostName=yourhub.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=Yj7RRPnkSDTv+UCFLgwIP/FrbDymZv4qVAIoTLHUFR8=');
```

X509 证书

使用 **OpenSSL** 作为 IOHandler

Certificate.Enabled：若身份验证使用证书，请启用此项。
Certificate.CertFile：X.509 客户端证书的路径。
Certificate.KeyFile：X.509 客户端密钥文件的路径。
Certificate.Password：如果证书设有密码，请在此填写。
Version：TLS 版本，默认使用 TLS 1.0

将 **SChannel** 用作 IOHandler

Certificate.Enabled：若身份验证使用证书，请启用此项。
Certificate.CertFile：PFX 证书路径（证书必须先转换为 PFX 格式）。[了解更多](#)。
Certificate.Password：如果证书设有密码，请在此填写。
Version：TLS 版本，默认使用 TLS 1.0

其他属性：

MQTTHeartBeat：若启用，尝试通过每 x 秒发送一次 ping 来保持 MQTT 连接活跃。

Interval：每次 ping 之间的秒数。

WatchDog：如果启用，当检测到意外断开连接时，将尝试自动重新连接到服务器。

Interval：两次重连尝试之间的秒数。

Attempts：最大重连尝试次数；零表示无限制。

LogFile：若启用，将套接字消息保存到日志文件（对调试有用）。如果从多个线程访问，日志文件的访问不是线程安全的。

Enabled：如果启用，每次套接字接收和发送消息时，都会将其保存到文件中。

FileName：文件的完整路径。

Azure MQTT 实现基于 MQTT 3.1.1 版本，但与规范存在以下偏差：

- IoT Hub 不支持 QoS 2 消息。如果设备应用以 QoS 2 发布消息，IoT Hub 将关闭网络连接。
- IoT Hub 不持久化 Retain 消息。如果设备发送了 RETAIN 标志设置为 1 的消息，IoT Hub 会在消息中添加 x-opt-retain 应用属性并将其传递给后端应用。
- IoT Hub 每台设备只支持一个活跃的 MQTT 连接。使用相同设备 ID 发起的任何新 MQTT 连接都会导致 IoT Hub 断开现有连接。

连接到 Azure IoT Hub

首先，您需要登录 Azure 账户、注册新设备，并为该设备创建认证方法。完成后，您可以创建一个新的 `TsgIoTAzure_MQTT_Client` 并连接到 Azure IoT Hub。

例如：

```
TsgIoTAzure_MQTT_Client *oClient = new TsgIoTAzure_MQTT_Client();
oClient->Azure->IoTHub = "youriothub.azure-devices.net";
oClient->Azure->DeviceId = "YourDeviceId";
oClient->SAS->Enabled = true;
oClient->SAS->SecretKey = "YourSecretKey";
oClient->OnMQTTConnect = OnMQTTConnectEvent;
oClient->Active = true;

private void OnMQTTConnect(TsgWSConnection *Connection, const bool Session, const TmqttConnReturnCode *ReturnCode)
{
    ShowMessage("Connected to Azure IoT Hub");
}
```

设备到云

从设备应用向解决方案后端发送信息时，IoT Hub 提供以下选项：

1. 设备到云消息，用于时序遥测和警报。

```
oClient->Send_DeviceToCloud("{\"temp\": 10}", azuIoTQoS1);
```

您可以使用 TStringList 发送键值属性，只需用所需的消息属性填充 TStringList 并将其作为参数传入即可。

```
TStringList *oProperties = new TStringList;
try
{
    oProperties->AddPair("prop_name1", "prop_value1");
    oProperties->AddPair("prop_name2", "prop_value2");
    oClient->Send_DeviceToCloud("{\"temp\": 10}", oProperties, azuIoTQoS1);
}
__finally
{
    oProperties->Free();
}
```

如果需要设置消息的 **ContentType** 和 **ContentEncoding**，必须将这些值添加到属性列表中。这些属性的名称由 Azure 定义。

Name	值
\$.ct	application/json
\$.ce	utf-8

```
TStringList *oProperties = new TStringList();
try
{
    oProperties->AddPair("$.ct", "application/json");
    oProperties->AddPair("$.ce", "utf-8");
    oClient->Send_DeviceToCloud("{\"temp\": 10}", oProperties, azuIoTQoS1);
}
__finally
{
    oProperties->Free();
}
```

2. 设备孪生的报告属性，用于报告设备状态信息，例如可用功能、条件或长期运行工作流的状态。例如，配置和软件更新。

```
oClient->Set_DeviceTwinsProperties("1", "{\"sgc\":1}");
```

云端到设备

IoT Hub 为设备应用提供三种向后端应用暴露功能的选项：

1. 直接方法用于需要立即确认结果的通信。直接方法通常用于设备的交互式控制，例如打开风扇。

```
oClient->Subscribe_DirectMethod();
```

您可以使用以下方法响应公共方法。

```
oClient->RespondPublicMethod(RequestId, Status, "Your Response", azuIoTQoS1);
```

2. Twin 的期望属性，用于旨在将设备置于某种期望状态的长时间运行命令。例如，将遥测发送间隔设置为 30 分钟。您可以使用以下方法获取属性。

```
oClient->Get_DeviceTwinsProperties("1");
```

3. **云到设备消息**，用于向设备应用发送单向通知。要接收消息，您必须先订阅。

```
oClient->Subscribe_CloudToDevice;
```

消息通过 **OnMQTTPublish** 事件（文本载荷）或 **OnMQTTPublishEx** 事件（载荷为 TsgcWSMQTTPublishData，包含 Value、Bytes 和 Stream 属性）接收。

```
void AzureIoTMQTTPublish(TsgcWSConnection *Connection, string aTopic, string aText)
{
    DoLog("Received Message: " + aTopic + " " + aText);
}
```

上传文件

IoT 中心通过提供共享访问签名（SAS）URI 或 X509 证书，帮助已连接设备上传文件。

如果您选择 SAS，必须设置以下属性：

- **Azure.IoTHub**：示例：youriothub.azure-devices.net
- **Azure.Deviceld**：示例：myDevice
- **SAS.SecretKey**：示例：Yj7RRPnkSDTv+UCFLgwIP/FrbDymZv4qVAlotLHUFR8=
- **SAS.KeyName**：示例：iothubowner
- **SAS.Enabled**：该值必须设置为 true。

如果您选择 X509 证书，必须设置以下属性：

- **Certificate.CertFile**：您的 PEM 证书路径。
- **Certificate.KeyFile**：您的 PEM 密钥文件路径。
- **Certificate.Enabled**：值必须设置为 true。

使用 **UploadFile** 方法将文件上传到 Azure 服务器。如果 Overwrite 参数设置为 true，将替换现有文件。如果 Overwrite 参数设置为 false，则仅在文件不存在时才上传，若文件已存在则引发错误（这是默认选项）。

```
void UploadFileToAzure()
{
    TOpenDialog* oDialog = new TOpenDialog(NULL);
    try
    {
        if (oDialog->Execute())
        {
            AnsiString fileName = oDialog->FileName;
            AzureIoT::UploadFile(fileName.c_str());
        }
    }
    __finally
    {
        oDialog->Free();
    }
}
```

Device Provisioning Service

Azure IoT 允许使用 DPS 从代码中注册设备。目前，该库支持通过传递 Scope Id 和 Registration Id 作为参数来注册设备。

```
TsgcIoTAzure_MQTT_Client* oClient = new TsgcIoTAzure_MQTT_Client(NULL);
try
{
    oClient->Certificate->CertFile = L"cert.pem";
    oClient->Certificate->KeyFile = L"key.pem";
    oClient->Certificate->Enabled = true;
    TsgcIoT_Azure_OperationRegistrationState* oResponse = new TsgcIoT_Azure_OperationRegistrationState();
```

```
try
{
    if (oClient->ProvisioningDeviceClient_Register(L"scope_id", L"registration_id", oResponse))
        ShowMessage(L"#Provisioning Register OK: " + oResponse->Status);
    else
        ShowMessage(L"#Provisioning Register Error: " + oResponse->Status);
}
__finally
{
    delete oResponse;
}
}
__finally
{
    delete oClient;
}
```

Azure IoT Explorer

您可以使用 **Azure IoT Explorer** 应用程序与连接到您的 IoT Hub 的设备进行交互，查看收到的遥测消息、已注册的设备等信息。该应用程序免费，可从以下地址下载：

<https://github.com/Azure/azure-iot-explorer/releases>

HTTP

HTTP 协议允许您从服务器获取资源，例如图像和 HTML 文档。它是一种客户端-服务器协议，这意味着客户端从服务器请求所需的资源。

当客户端希望连接到服务器时，需执行以下步骤：

1. 打开一个新的 TCP 连接
2. 向服务器发送包含请求数据的消息

```
GET / HTTP/1.1
Host: server.com
Accept-Language: en-us
```

3. 读取服务器发送的响应

```
HTTP/1.1 200 OK
Server: Apache
Content-Length: 120
Content-Type: text/html
...
```

组件

- **HTTP/2**：HTTP/2（或 h2）是一种二进制协议，为 Web 带来了推送、多路复用流和帧控制。
- **HTTP/1 客户端**：一个继承自 TIdHTTP 客户端组件的非可视组件。
- **OAuth2**：OAuth2 允许第三方应用程序对 HTTP 服务获得有限访问权限。
- **JWT**：JWT 允许创建带有可选签名和/或加密的数据，其负载保存有断言若干声明的 JSON。
- **Amazon SQS**：一种完全托管的消息队列服务，适用于微服务、分布式系统和无服务器应用程序。
- **Google Cloud Pub/Sub**：提供应用程序之间的消息传递，专为实现可靠的多对多异步消息传递而设计。
- **Google Calendar**：允许使用 Google Calendar API V3：获取日历、事件，与您自己的日历同步……
- **Google FCM**：使用 Firebase Cloud Messaging 发送通知。

HTTP/2

HTTP/2 是 HTTP 1.1 协议的演进，主要旨在提高网络使用效率。其语义保持不变，因此设计上与旧协议兼容。

HTTP 1.1 限制

HTTP 1.1 限于每个连接处理一个请求，因此客户端通常使用多个连接向服务器请求文件。但这带来了一个问题：当 TCP 连接过多时，客户端之间会竞争使用服务器资源，并且随着更多客户端连接到服务器，性能越来越低。

主要功能

- HTTP/2 是二进制协议（请记住，HTTP 1.1 是文本协议）。
- HTTP/2 通过 TLS 和 ALPN 工作。
- 它是多路复用的（允许您通过单个 TCP 连接发送多个请求）。
- 服务器可以向客户端推送响应。
- 减少往返时间，使客户端加载更快。

HTTP/2 还引入了其他改进，详见：[RFC7540](#)。

HTTP/2 需要我们的自定义 Indy 版本，因为它需要 ALPN 协议。

组件

- **TsgcHTTP2Client**：完全支持 HTTP/2 协议的客户端组件（sgcWebSockets 100% Pascal 代码，无需外部库）。
- **TsgcWebSocketHTTPServer**：完全支持 HTTP/2 协议的服务器组件（sgcWebSockets 100% Pascal 代码，无需外部库）。默认情况下 HTTP/2 已禁用，您可以通过 HTTPOptions 属性启用并将 Enable 设置为 true。
- **TsgcWebSocketServer_HTTPAPI**：支持 HTTP/2 协议的服务器组件（Microsoft 实现，需要 Windows 2016 及以上版本或 Windows 10 及以上版本）。
- **DataSnap 服务器**：DataSnap 服务器也可以支持 HTTP/2 协议。

API

- **Apple 推送通知**：从服务器提供商向用户设备推送面向用户的通知。

TsgcHTTP2Client

TsgcHTTP2Client 实现了客户端 HTTP/2 组件，可连接到 HTTP/2 服务器。

简介

TsgcHTTP2Client 实现了客户端 HTTP/2 组件，可以连接到 HTTP/2 服务器。请按照以下步骤配置此组件：

1. 创建一个新的 **TsgcHTTP2Client** 组件实例。
2. 向服务器发送请求，并使用 OnHTTP2Response 事件处理响应。示例：

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2Response = OnHTTP2ResponseEvent;
oClient->Get("https://www.google.com");

void OnHTTP2ResponseEvent(TObject *Sender, const
    TsgcHTTP2ConnectionClient *Connection, const TsgcHTTP2RequestProperty *Request,
    const TsgcHTTP2ResponseProperty *Response)
{
    ShowMessage(Response->DataString);
}
```

参考

- 属性 — 组件上提供 19 个属性。
- 方法 — 19 个公共方法。
- 事件 — 13 个可处理的事件。

指南

- 请求
 - [HTTP/2 请求方法](#)
 - [HTTP/2 服务器推送](#)
 - [下载文件](#)
 - [HTTP/2 部分响应](#)
 - [HTTP/2 头部](#)
- 连接
 - [客户端关闭连接](#)
 - [客户端保持连接活跃](#)
 - [HTTP/2 断连原因](#)
 - [客户端待处理请求](#)
- 身份验证
 - [客户端身份验证](#)
 - [HTTP/2 与 OAuth2](#)
- 类
 - [TsgcHTTP2ConnectionClient](#)
 - [TsgcHTTP2RequestProperty](#)
 - [TsgcHTTP2ResponseProperty](#)

TsgcHTTP2Client | 请求 HTTP/2 方法

HTTP/2 客户端可以工作在阻塞和非阻塞模式下。组件内部在辅助线程中工作，请求以异步方式处理，但您可以发起请求并等待其完成。

以下是一个示例，说明客户端如何向 HTTP/2 服务器请求 HTML 页面，以及如何在两种模式下工作。

异步模式

获取以下 URL：<https://www.google.com>，并在客户端接收完整响应时收到通知。调用 **GETASYNC** 方法后，处理过程继续执行，接收到响应时触发 OnHTTP2Response 事件。

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2Response = OnHTTP2ResponseEvent;
oClient->GetAsync("https://www.google.com");
void OnHTTP2ResponseEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection,
    const TsgcHTTP2RequestProperty *Request, const TsgcHTTP2ResponseProperty *Response)
{
    ShowMessage(Response->Headers->Text + #13#10 + Response->DataString);
}
```

阻塞模式

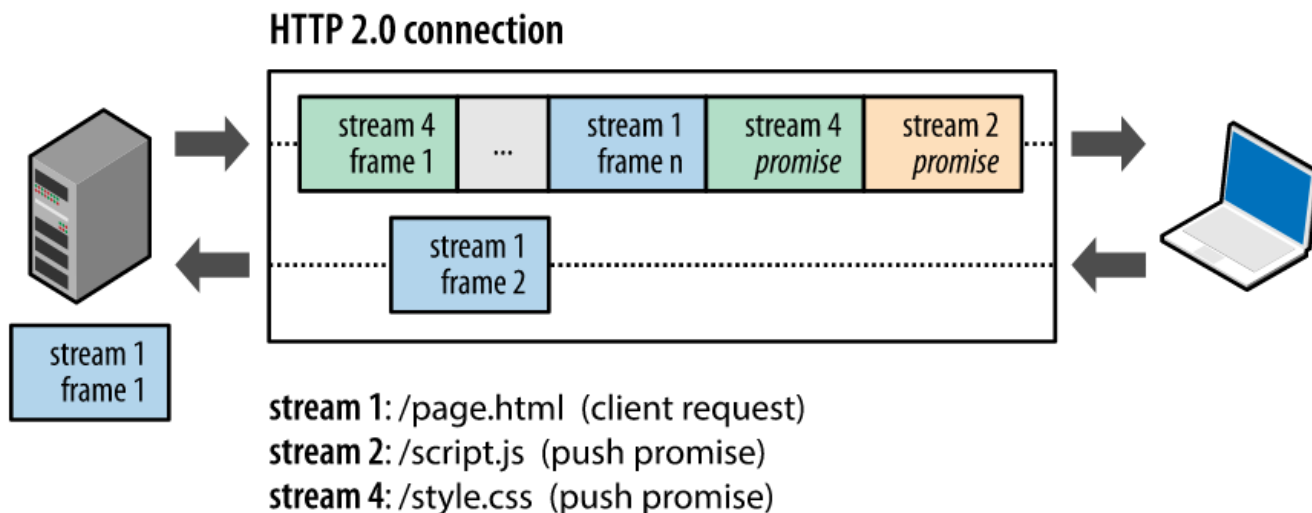
获取以下 URL：<https://www.google.com>，并等待客户端接收到完整响应。调用 **GET** 方法后，进程将等待直到收到响应或超时。

您可以通过 HTTP/2 客户端的 Response 属性访问原始响应数据，包括原始头、状态响应码、字符集等更多信息。

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
vResponse = oClient->Get("https://www.google.com");
if (oClient->Response->Status == 200)
    ShowMessage("Response from server: " + vResponse);
else
    ShowMessage("Response Code: " + IntToStr(oClient->Response->Status));
```

请求 | HTTP/2 服务器推送

服务器推送是服务器为单个客户端请求发送多个响应的能力。也就是说，除了对原始请求的响应外，服务器还可以向客户端推送额外资源，而无需客户端明确请求每个资源。



每次服务器向客户端发送 PushPromise 消息时，将调用 OnHTTP2PushPromise 事件。当客户端收到 PushPromise 时，表示服务器将在后续数据包中发送该资源，客户端可以接受或拒绝它。

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2PushPromise = OnHTTP2PushPromiseEvent;
oClient->Get("https://http2.golang.org/serverpush");
...
void OnHTTP2PushPromiseEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection,
    const TsgcHTTP2_Frame_PushPromise *PushPromise, bool &Cancel)
{
    if (PushPromise->URL == "/serverpush/static/godocs.js")
    {
        Cancel = true;
    }
    else
    {
        Cancel = false;
    }
}
```

TsgcHTTP2Client | HTTP/2 下载文件

当客户端向服务器请求文件时，使用 `OnHTTP2Response` 事件加载流响应。

大文件下载

下载大文件（数百 MB 或更大）时，将 `HTTP2Options.ReadTimeout` 设为 `0`（无超时），以确保传输不会被默认的 60 秒超时中断：

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->HTTP2Options->ReadTimeout = 0; // no timeout for large files
oClient->Get("https://server/largefile", oStream);
```

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2Response = OnHTTP2ResponseEvent;
oClient->Get("https://http2.golang.org/file/gopher.png");
...
void OnHTTP2ResponseEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection,
    const TsgcHTTP2RequestProperty *Request, const TsgcHTTP2ResponseProperty *Response)
{
    TFileStream *oStream = new TFileStream("file", fmOpenWrite | fmCreate);
    try
    {
        oStream->CopyFrom(Response->Data, Response->Data->Size);
    }
    __finally
    {
        oStream->Free();
    }
}
```

TsgcHTTP2Client | HTTP/2 部分响应

通常当您发送 HTTP 请求时，服务器会以所请求的文件进行响应；但有时服务器不是发送单个响应，而是像流一样发送多个响应，此时您可以使用 **OnHTTP2ResponseFragment** 事件捕获这些响应并向用户展示。

示例：向 <https://http2.golang.org/clockstream> 发送请求，服务器将每秒发送一次流式响应。

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2ResponseFragment = OnHTTP2ResponseFragmentEvent;
oClient->Get("https://http2.golang.org/clockstream");
...
void OnHTTP2ResponseFragmentEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection,
    const TsgcHTTP2RequestProperty *Request, const TsgcHTTP2ResponseFragmentProperty *Fragment)
{
    ShowMessage(Fragment->DataString);
}
```

TsgcHTTP2Client | HTTP/2 头部

TsgcHTTP2Client 允许自定义客户端连接时发送到服务器的头信息

示例：如果您需要添加 HTTP 请求头 "Client: sgcWebSockets"

```
void OnHTTP2BeforeRequest(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection,
    ref TStringList *Headers)
{
    Headers->Add("Client: sgcWebSockets");
}
```

您也可以使用 Request.CustomHeaders 添加自定义头信息。

TsgcHTTP2Client | 客户端关闭连接

可以使用 Active 属性或 Close/Disconnect 方法关闭连接。

Active 属性

当连接处于活跃状态时，若将 Active 设置为 False，连接将立即关闭，不会向服务器发送任何关于断开连接的消息。

断开连接

您可以使用 Disconnect 方法（来自 TsgcHTTP2Client 或 TsgcHTTP2ConnectionClient）断开套接字连接。

关闭

此方法向服务器发送消息，通知其连接即将关闭，并可选择性地发送断开原因的相关信息。这是关闭 HTTP/2 连接的一种干净方式。

Close 方法可以从 TsgcHTTP2Client 或 TsgcHTTP2ConnectionClient 对象调用。

可发送的错误原因如下：

- 无错误
- 协议错误
- 内部错误
- 流控错误
- settings timeout
- 流已关闭
- frame size error
- refused stream
- cancel
- 压缩错误
- 连接错误
- 请保持冷静
- 安全性不足
- required

TsgcHTTP2Client | 保持客户端连接活跃

客户端连接到服务器后，有时连接可能因信号不良、连接错误等而关闭。有两个属性可以帮助保持连接活跃。

HeartBeat

HeartBeat 属性允许您每隔 X 秒发送一次 **Ping**，以保持连接活跃。某些服务器会在对端之间没有数据交换时关闭 TCP 连接。HeartBeat 通过按固定间隔发送 ping 解决此问题，通常足以维持连接活跃。

属性 **HeartBeatType** 允许自定义心跳的工作方式：

1. **hbtAlways**：每隔 Interval 中定义的 x 秒发送一次 ping。
2. **hbtOnlyIfNoMsgRcvInterval**：仅在最近 Interval 属性定义的 x 秒内未收到任何消息时，每 x 秒发送一次 Ping。

示例：每 30 秒发送一次 ping

```
oClient = new TsgcHTTP2Client();
oClient->HeartBeat->Interval = 30;
oClient->HeartBeat->Enabled = true;
oClient->Active = true;
```

WatchDog

如果启用了 WatchDog，当客户端检测到断线时，WatchDog 将每隔 X 秒尝试重新连接，直到连接再次激活。

示例：断开连接后每隔 10 秒重新连接一次，尝试次数不限。

```
oClient = new TsgcHTTP2Client();
oClient->WatchDog->Interval = 10;
oClient->WatchDog->Attempts = 0;
oClient->WatchDog->Enabled = true;
oClient->Active = true;
```


TsgcHTTP2Client | HTTP/2 断开连接原因

HTTP/2 服务器可能因多种原因断开客户端连接，当服务器希望告知客户端断开原因时，会向客户端发送一条 **GoAway** 消息，其中包含断开信息。

使用 `OnHTTP2GoAway` 事件捕获服务器断开连接的原因（若客户端希望关闭连接，可使用 `close` 方法发送关闭原因）。

TsgcHTTP2GoAwayProperty 对象 包含断开连接的信息

- **LastStreamId**：是服务器处理的最后一个流。
- **ErrorCode**：标识错误代码的整数。
- **ErrorDescription**：错误描述，以下之一：
 - 无错误
 - 协议错误
 - 内部错误
 - 流控错误
 - settings timeout
 - 流已关闭
 - frame size error
 - refused stream
 - cancel
 - 压缩错误
 - 连接错误
 - 请保持冷静
 - 安全性不足
 - required
- **AdditionalDebugData**：可选字符串，提供有关断开连接的更多信息。

TsgcHTTP2Client | 客户端待处理请求

当客户端发送多个请求时，这些请求在辅助线程中处理，有时连接可能因某种原因关闭，此时仍有待处理的请求。使用 `OnHTTP2PendingRequests` 事件处理这些待处理请求。当客户端检测到断开连接且仍有待处理请求时，将调用此事件。此事件有 2 个参数：

1. **Reconnect**：默认禁用，如果设为 `True`，客户端将自动重新连接。
2. **Clear**：默认启用，若设置为 `false`，则客户端再次连接时将尝试重新向服务器发送待处理的请求。

TsgcHTTP2Client | 客户端身份验证

HTTP/2 客户端支持 2 种身份验证类型：Basic 身份验证和 OAuth2 身份验证。

使用 **OnHTTP2Authorization** 事件处理两种类型的身份验证。

基本认证

如果服务器返回请求基本认证的标头，请在 **OnHTTP2Authorization** 中设置用户名和密码。

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2Authorization = OnHTTP2AuthorizationEvent;
...
void OnHTTP2AuthorizationEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection, const string AuthType)
{
    if (AuthType == "Basic")
    {
        UserName = "user";
        Password = "secret";
    }
}
```

Bearer Token

如果服务器返回请求 Bearer 令牌身份验证的头部，请在 **OnHTTP2Authorization** 中设置令牌。

```
TsgcHTTP2Client *oClient = new TsgcHTTP2Client();
oClient->OnHTTP2Authorization = OnHTTP2AuthorizationEvent;
...
void OnHTTP2AuthorizationEvent(TObject *Sender, const TsgcHTTP2ConnectionClient *Connection, const string AuthType)
{
    if (AuthType == "Bearer")
    {
        aToken = "bearer token";
    }
}
```

来自第三方的 Bearer 值

如果您已经知道 Bearer 值（因为您通过其他方式获得了它），可以使用请求的以下属性将 Bearer 值作为 HTTP 标头传递，只需在调用任何 HTTP 请求方法之前设置：

```
TsgcHTTP2Client.Request.BearerAuthentication
= true

TsgcHTTP2Client.Request.BearerToken = "< value of the token >"
```

OAuth2

如果您想将 [OAuth2 组件](#) 与 [HTTP/2 客户端](#) 配合使用，请阅读以下文章。

TsgcHTTP2Client | HTTP/2 与 OAuth2

OAuth2 是 Google 等多家公司使用的常见授权方法。当您想要针对 Google 服务器进行身份验证以使用其任何 API 时，通常需要使用 OAuth2 进行身份验证。

sgcWebSockets 在 HTTP/2 客户端下支持 OAuth2，有一个名为 **Authentication.Token.OAuth** 的属性，您必须将 [TsgcHTTP_OAuth2](#) 的实例分配给它。

如何连接到 GMail Google API

为了连接到 Google API，我们需要创建一个 TsgcHTTP_OAuth2 的实例并填写以下数据：

```
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';
```

填写完 OAuth2 客户端组件后，创建一个新的 TsgcHTTP2Client 实例，并将 OAuth2 组件赋值给 HTTP/2 客户端。

```
TsgcHTTP2Client1.Authentication.Token.OAuth := TsgcHTTP_OAuth1;
```

最后，发起请求以获取账户 yourname@gmail.com 的消息列表。

```
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 是客户端 HTTP/2 连接的封装，您可以在 Client 事件中访问此对象。

方法

- **Ping**：向服务器发送 ping 以维持连接活跃。
- **Close**：向服务器发送消息，说明断开连接的原因。
- **Disconnect**：关闭连接而不向服务器发送任何信息消息。
- **HTTP/2 方法**：

GET：GET 方法请求指定资源的表示形式。使用 GET 的请求应仅用于检索数据。

HEAD：HEAD 方法请求与 GET 请求相同的响应，但不含响应正文。

POST：POST 方法用于将实体提交到指定资源，通常会导致服务器上的状态更改或副作用。

PUT：PUT 方法用请求有效载荷替换目标资源的所有当前表示。

DELETE：HEAD 方法要求的响应与 GET 请求的响应相同，但不包含响应体。

CONNECT：CONNECT 方法建立到目标资源所标识的服务器的隧道。

OPTIONS：OPTIONS 方法用于描述目标资源的通信选项。

TRACE：TRACE 方法沿到达目标资源的路径执行消息回路测试。

PATCH：PATCH 方法用于对资源应用部分修改。

TsgcHTTP2RequestProperty

此对象作为 OnHTTP2Response 事件的参数接收，用于了解服务器发送的响应所对应的原始请求。

属性

- **Method**：标识 HTTP/2 方法（GET、POST...）
- **URL**：所请求的 URL。
- **Request**：包含请求的字段。

TsgcHTTP2ResponseProperty

此对象作为参数在 OnHTTP2Response 事件中接收，它允许了解服务器发送给客户端的响应。

属性

- **Headers:** 包含从服务器接收到的原始请求头列表。
- **Data :** 包含服务器响应请求时发送的原始主体。
- **DataString :** 将 Data 转换为字符串的结果。
- **DataUTF8 :** Data 转换为 UTF8 字符串。
- **PushPromise :** 若已赋值，则包含服务器发送给客户端的 PushPromise 对象（表示此响应对象并非由客户端请求）。

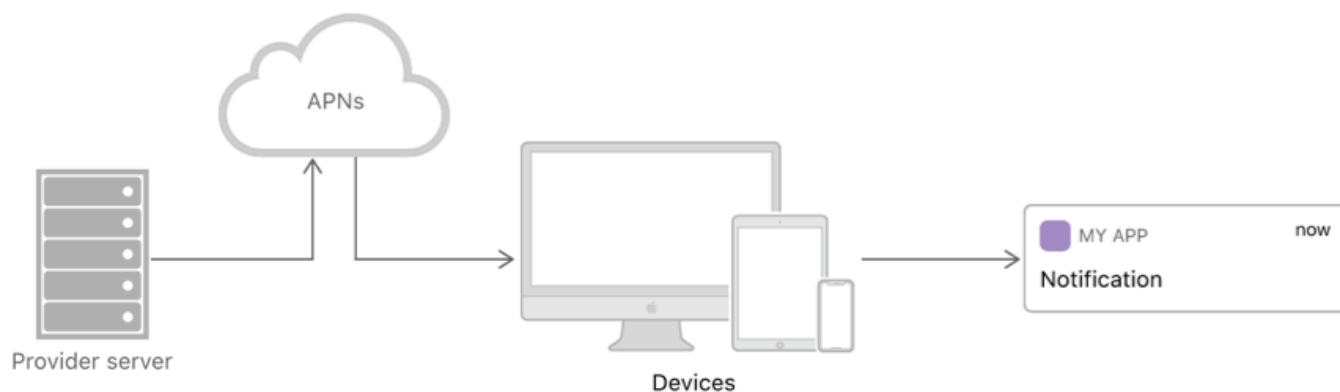
HTTP2 | 苹果推送通知

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server

Apple 允许使用 Apple 推送通知服务（APNs）向 Apple 设备发送推送通知。

当您想向设备发送通知时，提供商必须向 APNs 发送包含以下信息的 HTTP/2 POST 请求：

- 包含您要发送信息的 JSON 载荷。
- 标识用户设备的设备令牌。
- 关于如何投递通知的一些 HTTP 标头。
- 用于向 APNs 验证请求的 SSL 证书或 JWT Token



发送通知的必要条件

要使用 Rad Studio 向您的设备发送通知，您必须按照以下步骤操作

- 向 APNs 注册您的 APP
- [生成远程通知](#)
- [向 APNs 发送通知请求](#)
 - [基于令牌的 APNs 连接](#)
 - [基于证书的 APNs 连接](#)

APN | 生成远程通知 APNs

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/generating_a_remote_notification

Apple 通知使用 JSON 有效载荷发送通知对象。有效载荷的最大大小为 4096 字节。

JSON 载荷示例

简单提醒消息

```
{
  "aps": {
    "alert": "Alert from sgcWebSockets!"
  }
}
```

带标题和副标题的警告。

```
{
  "aps" : {
    "alert" : {
      "title" : "Game Request",
      "subtitle" : "Five Card Draw",
      "body" : "Bob wants to play poker",
    },
    "category" : "GAME_INVITATION"
  },
  "gameID" : "12345678"
}
```

APN | 向 APNs 发送通知请求

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server

将您的远程通知有效载荷和设备令牌信息发送到 Apple 推送通知服务（APNs）。

如何连接到 APNs

您必须使用 HTTP/2 协议且至少使用 TLS 1.2 或更高版本，才能在您的服务提供商与以下任一服务器之间建立成功连接：

开发服务器：https://api.sandbox.push.apple

生产服务器：https://api.push.apple

示例代码

创建一个新的 TsgcHTTP2Client 实例，并调用 POST 方法向 APNs 发送通知。

```
TsgcHTTP2Client *oHTTP = new TsgcHTTP2Client();
try
{
    // ... requires authorization code
    TStringStream *oStream = new TStringStream("{\"aps\":{\"alert\":\"Alert from sgcWebSockets!\"}}",
        TEncoding::UTF8);
    try
    {
        oHTTP->Post("https://api.push.apple/3/device/device_token", oStream);
        if (oHTTP->Response->Status == 200)
        {
            ShowMessage("Notification Sent Successfully");
        }
        else
        {
            ShowMessage("Notification error");
        }
    }
    __finally
    {
        oHTTP->Free();
    }
}
__finally
{
    oHTTP->Free();
}
```

要发送通知，您必须使用 HTTP/2 协议和 TLS 1.2 或更高版本，与 APNs 建立基于令牌或基于证书的信任。

- [基于令牌的 APNs 连接](#)
- [基于证书连接到 APNs](#)

APNs 受信 | 基于令牌的 APNs 连接

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/establishing_a_token-based_connection_to_apns

使用无状态身份验证令牌保护与 Apple Push Notification service (APNs) 的通信。

首先，您必须从 Apple Developer 账户获取**加密密钥**和**密钥 ID**。成功注册后，您将获得一个 10 字符的密钥 ID 字符串，以及一个 .p8 扩展名的身份验证令牌签名密钥文件。

您必须使用 `sgcWebSockets` [JWT 客户端](#)，以 **ES256** 算法生成 JWT。令牌不必在每次 HTTP/2 请求时重新生成，且令牌的刷新时间不得早于 20 分钟，也不得晚于 60 分钟。

配置 JWT 客户端

使用以下值配置 JWT 客户端：

- **JWTOptions.Header.Algorithm**：是您用于加密令牌的加密算法。APNs 仅支持 ES256 算法。
- **JWTOptions.Header.kid**：从您的开发者账户获取的 10 字符密钥 ID。
- **JWTOptions.Payload.iss**：该值为您用于开发公司应用程序的 10 字符 Team ID。请从您的开发者账户中获取此值。
- **JWTOptions.Payload.iat**：“签发时间” (issued at)，其值表示生成该 JSON 令牌的时间。请以自 Epoch (UTC) 起的秒数指定该值。该值与当前时间之差不得超过一小时。
- **JWTOptions.RefreshTokenAfter**：将值设置为 40 分钟 (60*40) 秒。

使用基于令牌的连接需要发送 **apns-topic**，其值为您应用的 bundle ID/app ID (例如：com.example.application)。

```
TsgcHTTP2Client *oHTTP = new TsgcHTTP2Client(NULL);
oHTTP->TLSOptions->IOHandler = iohOpenSSL;

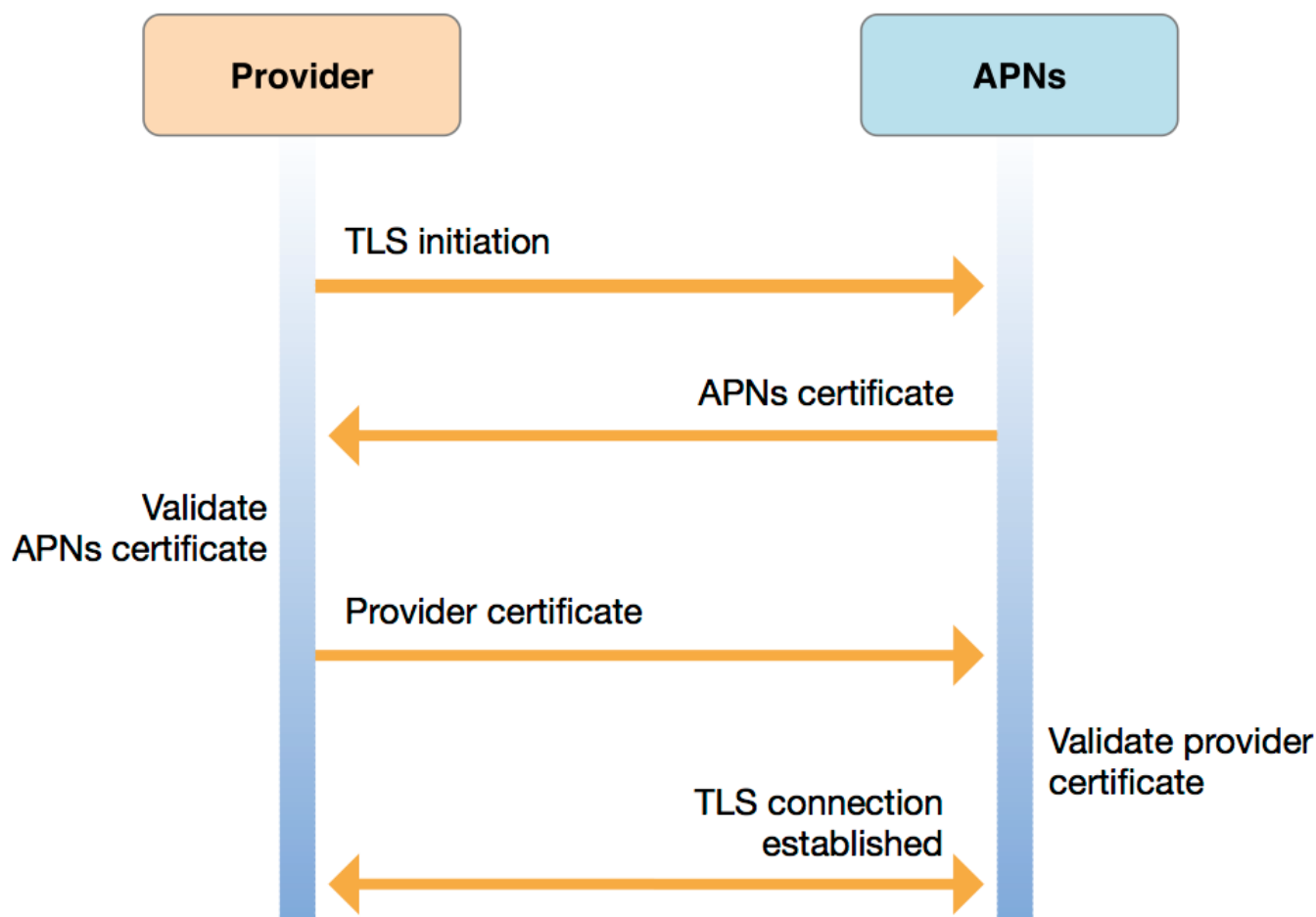
TsgcHTTP_JWT_Client *oJWT = new TsgcHTTP_JWT_Client(NULL);
oHTTP->Authentication->Token->JWT = oJWT;
oJWT->JWTOptions->Header->alg = jwtES256;
oJWT->JWTOptions->Header->kid = "apple key id";
oJWT->JWTOptions->Payload->iss = "issuer";
oJWT->JWTOptions->Payload->iat = StrToInt64(GetDateTimeUnix(Now, False));
oJWT->JWTOptions->Algorithms->ES->PrivateKey->LoadFromFile("AuthKey_*.p8");
oJWT->JWTOptions->RefreshTokenAfter = 60*40;

oHTTP->Request->CustomHeaders->Clear();
oHTTP->Request->CustomHeaders->Add("apns-topic: com.example.application");
```

基于证书的 APNs 连接

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/establishing_a_certificate-based_connection_to_apns

您可以使用从 Apple 获取的证书来保护与 Apple 推送通知服务 (APNs) 的通信。



首先进入您的开发者账号，并为 Apple Push Notification 服务**创建新证书**

下载证书后，sgcWebSockets HTTP/2 客户端支持 2 种安全 IOHandler（仅限 Windows；其他平台仅支持 OpenSSL）。

- OpenSSL
- SChannel（仅限 Windows）

OpenSSL

若使用 OpenSSL，必须将 OpenSSL 库与应用程序一同部署。在使用 [TsgcHTTP2Client](#) 设置证书之前，必须先将该证书转换为 PEM 格式，因为 OpenSSL 不支持直接导入 P12 证书。

使用以下命令将单个 P12 证书转换为 PEM 格式的证书和私钥文件

创建 PEM 证书文件

```
openssl pkcs12 -in INFILE.p12 -out OUTFILE.crt -nokeys
```

创建私钥文件

```
openssl pkcs12 -in INFILE.p12 -out OUTFILE.key -nodes -nocerts
```

一旦您拥有 PEM 格式的证书和私钥，可以按如下方式配置 [TsgcHTTP2Client](#)。

```
TsgcHTTP2Client *oHTTP = new TsgcHTTP2Client(NULL);
oHTTP->TLSOptions->IOHandler = iohOpenSSL;
oHTTP->TLSOptions->CertFile = "certificate_file.pem";
oHTTP->TLSOptions->KeyFile = "private_key.pem";
oHTTP->TLSOptions->Password = "certificate password";
oHTTP->TLSOptions->Version = tls1_2;
```

SChannel

如果您使用 SChannel，则无需部署任何库，从 Apple 下载的证书可以直接导入，无需事先转换为 PEM 格式。使用 SChannel 作为 IOHandler 时，请将 UseLegacyCredentials 属性设置为 True。

```
TsgcHTTP2Client *oHTTP = new TsgcHTTP2Client(NULL);
oHTTP->TLSOptions->IOHandler = iohSChannel;
oHTTP->TLSOptions->SChannel_Options->UseLegacyCredentials = true;
oHTTP->TLSOptions->CertFile = "certificate_file.p12";
oHTTP->TLSOptions->Password = "certificate password";
oHTTP->TLSOptions->Version = tls1_2;
```

错误

如果您收到错误 **"missing topic"**，您很可能正在使用通用证书（可用于推送通知、VoIP 等的证书），需要将主题名称设置为您应用程序的 bundle ID/app id（例如：com.example.application）。只需在 HTTP/2 客户端的 Request 属性中将 apns-topic 标头设置为正确的值即可。

```
oHTTP->Request->CustomHeaders->Clear();
oHTTP->Request->CustomHeaders->Add("apns-topic: com.example.application");
```

HTTP/1

TsgcHTTP1Client 是一个非可视组件，继承自 **TIdHTTP Indy** 组件并添加了一些新属性。此组件位于 **sgcHTTP** 单元中。

TLSOptions

允许您配置如何使用 HTTP/1 协议连接到安全 SSL/TLS 服务器。

ALPNProtocols：将发送给服务器的 ALPN 协议列表。

RootCertFile：根证书文件的路径。

CertFile：证书文件的路径。

KeyFile：证书密钥文件的路径。

Password：如果证书使用密码保护，请在此处设置密码。

VerifyCertificate：如果需要验证证书，请启用此属性。使用 **OnSSLVerifyPeer** 事件自定义 SSL 验证。

VerifyDepth：一个整数属性，表示对 X.509 证书执行验证时允许的最大链路数。

Version：默认使用 TLS 1.0。若服务器要求更高的 TLS 版本，可在此处选择。

代理：在此处您可以定义是否要通过代理服务器连接，可以连接到以下代理服务器：

pxyHTTP：HTTP 代理服务器。

pxySocks4：SOCKS4 代理服务器。

pxySocks4A：SOCKS4A 代理服务器。

pxySocks5：SOCKS5 代理服务器。

IOHandler：选择使用哪个库通过 TLS 进行连接。

iohOpenSSL：使用 OpenSSL 库，是 Indy 组件的默认选项。需要为 win32/win64 部署 OpenSSL 库。

iohSChannel：使用 Secure Channel，这是 Microsoft 为 Windows 实现的安全协议。无需部署 OpenSSL 库。仅适用于 Windows 32/64 位。

OpenSSL_Options：OpenSSL 库的配置。

APIVersion：允许定义将使用哪个 OpenSSL API 版本。

oslAPI_1_0：使用 API 1.0 OpenSSL，这是 Indy 支持的最新版本

oslAPI_1_1：使用 OpenSSL API 1.1，需要我们定制的 Indy 库，并支持 OpenSSL 1.1.1 库（含 TLS 1.3 支持）。

oslAPI_3_0：使用 API 3.0 OpenSSL，需要我们的自定义 Indy 库，并允许使用 OpenSSL 3.0.0 库（支持 TLS 1.3）。

LibPath：在此配置 OpenSSL 库的所在路径

oslpNone：这是默认值。OpenSSL 库应位于与二进制文件相同的文件夹或已知路径中。

oslpDefaultFolder：自动为所有 IDE 类型设置 OpenSSL 库应所在的默认路径。

oslpCustomFolder：如果选择此选项，请在属性 **LibPathCustom** 中定义完整路径。

LibPathCustom：当 **LibPath** = **oslpCustomFolder** 时，在此定义 OpenSSL 库所在的完整路径。

UnixSymLinks：启用或禁用 Unix 系统下加载符号链接（默认启用，OSX64 除外）：

oslsSymLinksDefault：默认情况下，符号链接是启用的，但 OSX64 除外（macOS Monterey 及更高版本在尝试加载不带版本号的库时失败）。

oslsSymLinksLoadFirst：优先加载符号链接，然后再尝试加载带版本号的库。

oslsSymLinksLoad：在尝试加载带版本的库之后加载符号链接。

oslsSymLinksDontLoad：不加载符号链接。

MinVersion：在此设置客户端连接到安全服务器所使用的最低版本。默认值为 **tlsUndefined**，表示最低版本与 **Version** 属性中设置的版本相同。示例：如果您希望客户端仅使用 TLS 1.2 或 TLS 1.3 连接，请设置以下值。

SSLOptions.Version := **tls1_3**;

SSLOptions.OpenSSL_Options.MinVersion := **tls1_2**;

X509Checks：使用此属性启用额外的 X509 证书验证：

Mode：选择要验证的选项

oslx509chHostName：验证主机名证书。

oslx509chIPAddress：验证证书的 IP 地址。

HostName：如果主机名与请求不同，请设置主机名。

IPAddress：若 IP 地址与请求不同，请设置该 IP 地址。

SChannel_Options：允许您使用 Windows 证书存储中的证书。

CertHash：是证书哈希值。您可以在 PowerShell 中运行 **dir** 命令找到证书哈希值。

CipherList：在此设置将使用的密码套件（以":"分隔）。示例：CALG_AES_256:CALG_AES_128

CertStoreName：证书存储的存储名称。请从以下选项中选择：

- scsnMY**（默认值）
- scsnCA**
- scsnRoot**
- scsnTrust**

CertStorePath：存储证书的存储区路径。请选择以下之一：

- scspStoreCurrentUser**（默认值）
- scspStoreLocalMachine**

日志

如果启用了 Log 属性，它会将套接字消息保存到指定的日志文件中，便于调试。

LogOptions.FileName：文件的完整路径。

身份验证

允许您使用 [OAuth2](#) 或 [JWT](#) 进行身份验证。

异步请求

默认情况下，HTTP1Client 使用阻塞请求。或者，您可以使用异步方法在辅助线程中执行这些 HTTP 请求，避免阻塞调用请求的线程。实现了以下异步方法：

- GetAsync
- PostAsync
- PutAsync
- OptionsAsync
- DeleteAsync

调用这些方法后，代码不会等待响应，而是继续执行下一行，响应可通过事件 **OnAsyncResponse** 处理。

```
void __fastcall OnAsyncResultEvent(TObject* Sender, const TsgcHTTPAsyncRequest* aRequest, const TIdHTTPResponse*
```

如果在处理异步请求时发生任何错误，将在 **OnAsyncException** 事件中触发该异常。

示例

向 HTTPS 服务器请求 **GET** 方法，并使用 **TLS 1.2**

```
TsgcHTTP1Client *oHTTP = new TsgcHTTP1Client();
try
{
    oHTTP->TLSOptions->Version = tls1_2;
    ShowMessage(oHTTP->Get("https://www.google.es"));
}
__finally
{
    oHTTP->Free();
}
```

使用 **openssl 1.1** 和 **TLS 1.3** 向 HTTPs 服务器发送 **GET** 请求

```
TsgcHTTP1Client *oHTTP = new TsgcHTTP1Client();
try
```

```

{
    oHTTP->TLSOptions->OpenSSL_Options->APIVersion = sslAPI_1_1;
    oHTTP->TLSOptions->Version = tls1_3;
    ShowMessage(oHTTP->Get("https://www.google.es"));
}
__finally
{
    oHTTP->Free();
}

```

请求一个**异步 POST** 方法，并使用 **OnAsyncResultEvent** 读取响应。

```

void __fastcall OnAsyncExceptionEvent(TObject* Sender, const TsgcThread* aThread, const Exception* E)
{
    Log(E->Message);
}
void __fastcall OnAsyncResultEvent(TObject* Sender, const TsgcHTTPAsyncRequest* aRequest, const TIdHTTPResponse*
{
    if (aResponse->ResponseCode == 200)
        Log("ok", aRequest->Response);
    else
        Log("error", aRequest->Response);
}
TsgcHTTP1Client* oHTTP = new TsgcHTTP1Client(NULL);
oHTTP->OnAsyncResult = OnAsyncResultEvent;
oHTTP->OnAsyncException = OnAsyncExceptionEvent;
TStringStream* oRequest = new TStringStream("body");
TStringStream* oResponse = new TStringStream("");
oHTTP->PostAsync("https://localhost/test", oRequest, oResponse);

```

使用 **Windows** 的 **SChannel** 向 HTTPS 服务器发送 GET 请求。

```

TsgcHTTP1Client *oHTTP = new TsgcHTTP1Client();
try
{
    oHTTP->TLSOptions->IOHandler = iohSChannel;
    oHTTP->TLSOptions->Version = tls1_2;
    ShowMessage(oHTTP->Get("https://www.google.es"));
}
__finally
{
    oHTTP->Free();
}

```

请求 **SSE** 方法以获取数据事件

```

TsgcHTTP1Client *oHTTP = new TsgcHTTP1Client();
oHTTP->OnSSEMessage = OnSSEMessageEvent;
oHTTP->GetSSE("https://www.yoursite.com/sse");

void OnSSEMessageEvent(TObject *Sender, const string aMessage, bool &Cancel)
{
    ShowMessage(aMessage);
}

```

事件

OnSSEMessage

当接收到新的 SSE 消息时调用此事件。

OnSSLVerifyPeer

如果启用了证书验证，在此事件中您可以验证并决定是否接受服务器证书。

OnSSLGetHandler

在 SSL 处理程序创建之前触发此事件。您可以在这里创建自己的 SSL 处理程序（需要继承自 `TIdServerIOHandlerSSLBase` 或 `TIdIOHandlerSSLBase`），并设置所需属性。

OnSSLAfterCreateHandler

如果未创建自定义 SSL 对象，则使用 OpenSSL 处理程序创建默认对象。您可以访问 SSL 处理程序属性并根据需要修改它们。

OnAsyncResult

请求异步方法（使用 `GetAsync`、`PutAsync` 等方法）后调用该事件。使用 `Response` 参数获知请求结果。

OnAsyncException

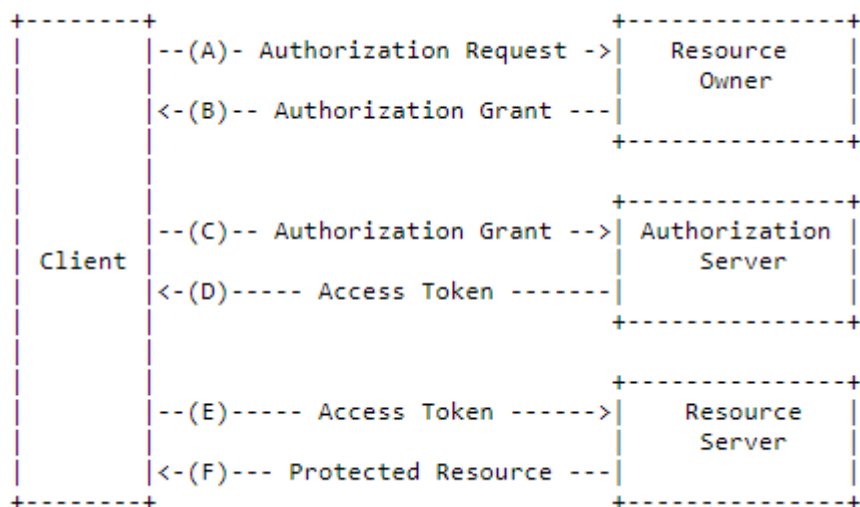
若在处理异步请求时发生任何错误，将触发此事件并传入所引发的异常。

HTTP | OAuth2

OAuth2 允许第三方应用程序代表资源所有者或以自身名义，获取对 HTTP 服务的有限访问权限。借助 OAuth2，服务提供商和消费方应用程序可以安全地相互交互。

在 OAuth2 中，共有 4 种角色：

- **Resource Owner**：用户。
- **资源服务器**：托管受保护资源并基于访问令牌提供访问权限的服务器。
- **Client**：寻求权限的外部应用程序。
- **Authorization Server（授权服务器）**：在对用户进行身份验证后签发访问令牌。



组件

- **TsgcHTTP_OAuth2_Client**：支持 OAuth2 的客户端，可以连接到 OAuth2 服务器（如 Google、Facebook 等）请求身份验证。
- **TsgcHTTP_OAuth2_Server**：是 OAuth2 协议的服务器端实现，允许您保护服务器资源。
- **TsgcHTTP_OAuth2_Server_Provider**：允许在您的服务器中实现外部 OAuth2 提供商（如 Azure AD、Google、Facebook 等），用户可使用 Azure、Google、Facebook 等账户凭据登录。

Server 和 **Client** OAuth2 组件支持 **PKCE**（授权码交换证明密钥），这是授权码流程的扩展，用于防止 CSRF 和授权码注入攻击（RFC 7636）。

TsgcHTTP_OAuth2_Client

当客户端需要新的访问令牌时，自动启动 HTTP 服务器以处理来自授权服务器的响应。

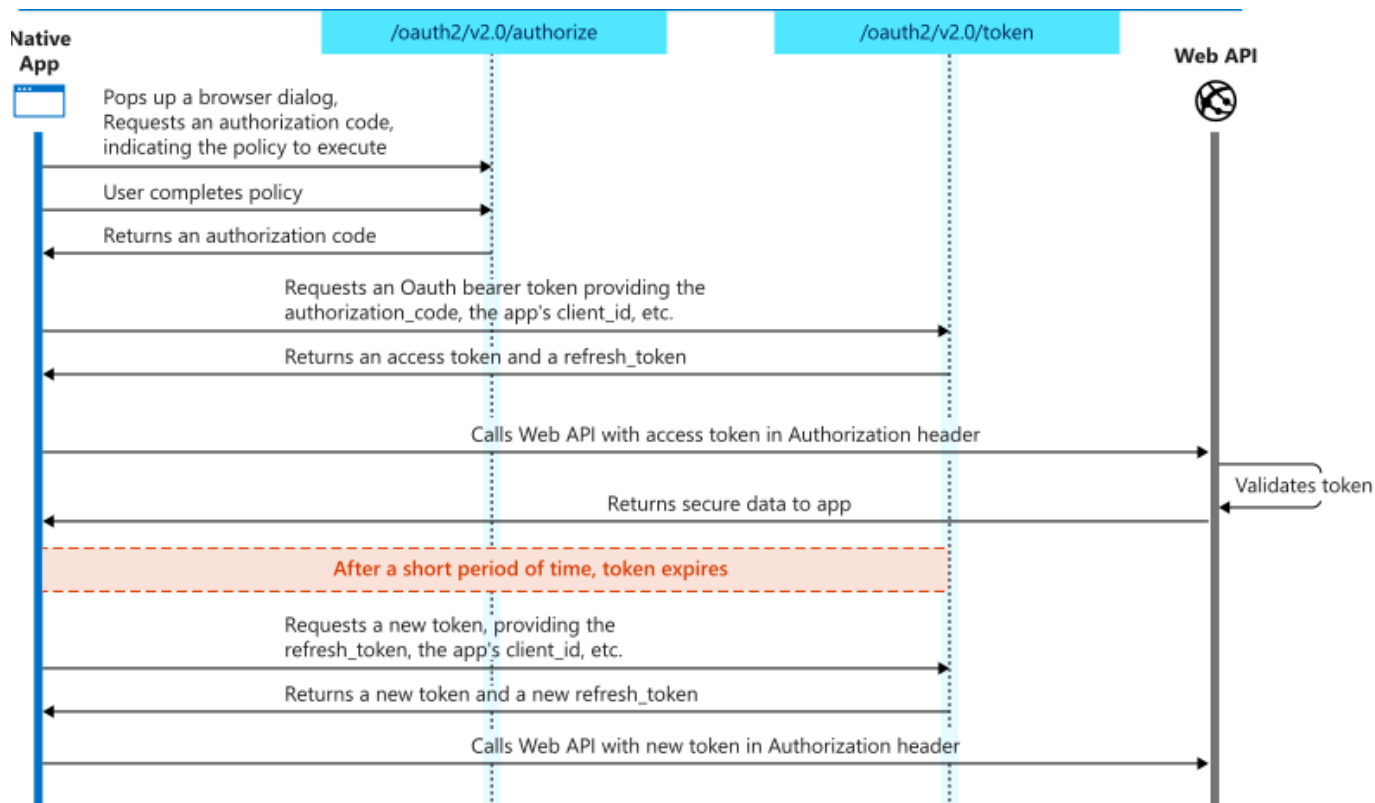
简介

此组件允许您处理客户端与其他角色之间的流程。基本上，当您设置 `Active := True` 时，会打开一个新的 Web 浏览器并请求用户授予授权；如果成功，授权服务器向应用程序发送一个令牌，组件对其进行处理，客户端随后可以使用该令牌连接到资源服务器。此组件启动一个简单的 HTTP 服务器，用于处理授权服务器响应，并使用 HTTP 客户端请求访问令牌。

GrantType

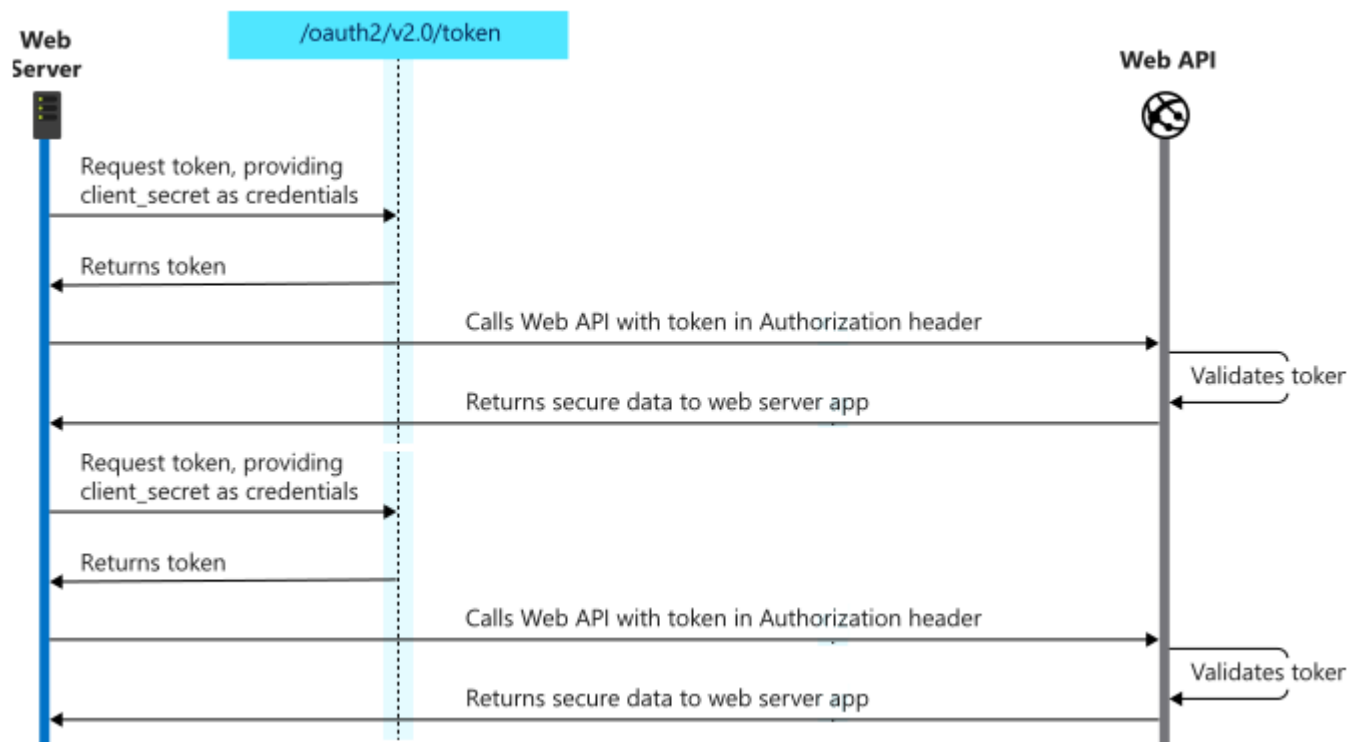
客户端支持以下授权类型：

auth2Code：用于在大多数应用程序类型（包括单页应用、Web 应用和原生安装应用）中执行身份验证和授权。该流程使应用能够安全获取可用于访问受保护资源的 `access_token`，以及用于获取额外 `access_token` 的刷新令牌，以及已登录用户的 ID 令牌。



auth2CodePKCE：与 `auth2Code` 身份验证流程相同，但启用了 PKCE。PKCE（代码交换证明密钥）是 OAuth 2.0 的安全扩展，旨在增强本机 and 单页应用程序授权流程的安全性。它降低了拦截攻击的风险，尤其是在公共客户端中，授权码在传输过程中可能面临被拦截的风险。通常，此选项用于本机应用和移动应用。

auth2ClientCredentials：此授权类型通常用于必须在后台运行且无需用户立即交互的服务器间交互。此类应用程序通常称为守护程序或服务账户。



auth2DeviceCode：设备授权授权（RFC 8628），适用于缺乏浏览器或输入能力有限的输入受限设备，如智能电视、媒体控制台和 IoT 设备。设备显示用户代码和验证 URI；用户在辅助设备（手机或电脑）上访问该 URI 并输入代码以完成授权。

LocalServerOptions

当客户端需要新的访问令牌时，自动启动一个 HTTP 服务器以处理来自授权服务器的响应。该服务器对用户透明，通常在 localhost 上运行。默认使用 8080 端口，如有需要可更改。

- **IP**：服务器监听 IP，示例：127.0.0.1
- **Port**：默认值为 8080。当 GrantType = auth2CodePKCE（用于桌面和原生应用程序）时，可以将端口值设置为零，服务器将选择一个随机端口。
- **RedirectURL**：（可选）允许自定义重定向 URL，示例：http://localhost:8080/oauth/。
- **SSL**：如果本地服务器运行在安全端口上，请启用此属性（*仅 Professional 和 Enterprise 版本支持）。
- **SSLOptions**：允许自定义服务器的 SSL 属性（*仅专业版和企业版支持）。
- **LogOptions**：允许您保存 HTTP 内部服务器接收和发送的请求/响应日志（*仅适用于 Professional 和 Enterprise 版本）。
 - **Enabled**：设置为 True 以后用文件日志记录。
 - **FileName**：设置存储日志文件的文件名。

AuthorizationServerOptions

在此处设置授权和访问令牌的 URL，通常由 API 规范提供。Scope 是客户端请求的所有范围的列表。示例：

- **AuthURL**：https://accounts.google.com/o/oauth2/auth
- **TokenURL**：https://accounts.google.com/o/oauth2/token
- **Scope**：https://mail.google.com/
- **RevocationURL**：令牌撤销端点的 URL（Revoke 方法必填）。示例：https://accounts.google.com/o/oauth2/revoke
- **IntrospectionURL**：令牌自省端点的 URL（Introspect 方法所需）。示例：https://accounts.google.com/o/oauth2/introspect

OAuth2Options

ClientId 是必填字段，用于告知服务器客户端的标识符。请查阅您的 API 规范以了解如何获取 ClientId。客户端密钥同理。

有时，服务器要求使用基本认证提供用户名和密码进行连接，如果是这种情况，您可以在 Username/Password 字段中进行设置。示例：

- **GrantType**：授权流程类型

- **auth2Code**：受信任的应用，如 Web 服务器。
- **auth2CodePKCE**：不受信任的原生或移动应用。
- **auth2ClientCredentials**：无需用户交互的自动化应用。
- **auth2ResourceOwnerPassword**：允许应用程序通过直接处理用户密码来登录用户。
- **auth2DeviceCode**：设备授权许可（RFC 8628），适用于无浏览器或输入能力有限的受限设备。
- **ClientId**：180803918307-eqjtm20gqfhcs6gjkbbrreng022mqqc.apps.googleusercontent.com
- **ClientSecret**：_by1iYYrvVHxC2Z8TbtNEYJN
- **Username**：
- **Password**：

HTTPClientOptions

在此处，您可以自定义连接到 HTTP 服务器以请求新令牌时的客户端选项。

TLSOptions：若启用 TLS，可在此处自定义部分 TLS 属性。

ALPNProtocols：将发送到服务器的 ALPN 协议列表。

RootCertFile：根证书文件的路径。

CertFile：证书文件的路径。

KeyFile：证书密钥文件的路径。

Password：如果证书有密码保护，请在此处设置。

VerifyCertificate：如果必须验证证书，请启用此属性。

VerifyDepth：一个整数属性，表示对 X.509 证书执行验证时允许的最大链路数。

Version：默认使用 TLS 1.0，如果服务器需要更高的 TLS 版本，可在此处选择。

IOHandler：选择使用哪个库通过 TLS 进行连接。

ioOpenSSL：使用 OpenSSL 库，是 Indy 组件的默认选项。需要为 win32/win64 部署 OpenSSL 库。

ioSChannel：使用由 Microsoft 为 Windows 实现的安全通道安全协议，无需部署 openssl 库。仅适用于 Windows 32/64 位。

OpenSSL_Options：允许定义将使用的 OpenSSL API。

APIVersion：允许定义将使用哪个 OpenSSL API 版本。

oslAPI_1_0：使用 API 1.0 OpenSSL，这是 Indy 支持的最新版本

oslAPI_1_1：使用 OpenSSL API 1.1，需要我们定制的 Indy 库，并支持 OpenSSL 1.1.1 库（含 TLS 1.3 支持）。

oslAPI_3_0：使用 API 3.0 OpenSSL，需要我们的自定义 Indy 库，并允许使用 OpenSSL 3.0.0 库（支持 TLS 1.3）。

LibPath：在此处您可以配置 openssl 库的位置

oslpNone：这是默认值，OpenSSL 库应与二进制文件位于同一文件夹中，或位于已知路径中。

oslpDefaultFolder：自动设置所有 IDE 版本应查找 OpenSSL 库的路径。

oslpCustomFolder：如果选择此选项，请在属性 LibPathCustom 中定义完整路径。

LibPathCustom：当 LibPath = oslpCustomFolder 时，在此处定义 openssl 库所在的完整路径。

UnixSymLinks：启用或禁用在 Unix 系统下加载符号链接（默认启用，OSX64 除外）：

oslsSymLinksDefault：默认情况下启用，但在 OSX64 下除外（MacOS Monterey 之后，不带版本号尝试加载库会失败）。

oslsSymLinksLoadFirst：先加载符号链接，然后再尝试加载版本库。

oslsSymLinksLoad：在尝试加载版本库后加载符号链接。

oslsSymLinksDontLoad：不加载符号链接。

SChannel_Options：允许您使用 Windows 证书存储中的证书。

CertHash：是证书哈希值。您可以在 PowerShell 中运行 dir 命令找到证书哈希值。

CertStoreName：存储证书的存储名称。从以下选项中选择：

scsnMY（默认值）

scsnCA

scsnRoot

scsnTrust

CertStorePath：存储证书的存储路径。请从以下选项中选择：

scspStoreCurrentUser（默认值）

scspStoreLocalMachine

LogOptions：如果设置了文件名，将保存 HTTP 客户端的 HTTP 请求/响应日志

参考

- 属性 — 组件上提供 6 个属性。
- 方法 — 7 个公共方法。
- 事件 — 可处理 19 个事件。

适用于 Web 应用程序的 OAuth2 客户端

当 OAuth2 客户端必须通过连接到 Web 应用程序来获取授权令牌时，用于获取授权令牌的本地服务器必须配置 Web 应用程序参数。请设置以下 **LocalServerOptions** 属性：

- **IP**：之前在 OAuth2 配置中配置的 IP 地址。
- **Port**：OAuth2 配置中之前配置的监听端口。
- **SSL**：如果服务器使用安全连接，请启用此选项。

桌面应用程序的 OAuth2 客户端

当 OAuth2 客户端是桌面或原生应用程序时，用于获取授权令牌的本地服务器可以监听本地 IP 地址（例如：127.0.0.1），端口可以随机选择。因此，应按如下方式配置 **LocalServerOptions**

- **IP**：127.0.0.1
- **Port**：设置为零，服务器将自动选择一个随机端口。

为桌面或原生应用程序创建 OAuth2 客户端时，请将 **OAuth2Options.GrantType** 设为 **auth2CodePKCE** 以添加额外安全性。

OAuth2 | TsgcHTTP_OAuth2_Client_Google

此组件让您可以轻松地使用 Google 账户登录。

配置

该模块首先需要 **在您的 Google 账户中配置您的 OAuth2 应用程序**，配置完成后，只需在您的应用程序中添加几行代码即可允许用户使用任意 Google 账户登录。

用于读取 Google 响应的 **本地服务器**，默认侦听 **IP 地址 127.0.0.1** 和 **端口 0**（随机端口）。因此，您必须在 Google 应用程序中配置 CallBack URL。当然，您也可以修改 IP 地址和端口。

在 Google 账号中配置好 OAuth2 应用程序后，只需创建一个 **TsgcHTTP_OAuth2_Client_Google** 并调用方法 **Authenticate**，将 **Client_Id** 和 **Client_Secret** 作为参数传入。该 **方法将等待**（默认最多 60 秒）直到用户 **成功登录**。返回一个对象，您可以通过它检查用户是否已通过身份验证、姓名、ID 以及用户配置文件中的更多数据。

示例

```
void GoogleSignIn()
{
    TsgcHTTP_OAuth2_Client_Google *oClient = new TsgcHTTP_OAuth2_Client_Google.Create(this);
    Try
    {
        TsgcOAuth2_Google_Data *oData = oClient.Authenticate("client_id", "client_secret");
        if oData->Authenticated() then
            ShowMessage(oData->UserProfile->_Name);
    }
    _Finally
    {
        oClient->Free();
    }
}
```

TsgcHTTP_OAuth2_Client_Microsoft

此组件允许您以简单的方式使用 Microsoft 账户登录。

配置

此模块首先需要在您的 Microsoft 账户中 **配置您的 OAuth2 应用程序**，配置完成后，只需在您的应用程序中添加几行代码，即可允许用户使用任意 Microsoft 账户登录。

用于读取 Microsoft 响应的 **本地服务器** 默认监听 **IP 地址 127.0.0.1** 和 **端口 8080**，并使用 SSL。因此，您必须在 Microsoft 应用程序中将回调 URL 配置为 **https://localhost:8080**（Microsoft 只允许将 localhost 作为本地 IP 地址）。当然，您也可以修改 IP 地址和端口。

在 Microsoft Account 中配置好 OAuth2 应用程序后，只需创建 **TsgcHTTP_OAuth2_Client_Microsoft** 的实例，并调用方法 **Authenticate**，将参数 **TenantId**、**Client_Id** 和 **Client_Secret** 传入。该 **方法将等待**（默认最多 60 秒），直到用户 **成功登录**。返回一个对象，您可以通过它检查用户是否已通过身份验证，以及用户名、ID 等更多用户资料数据。

示例

```
void GoogleSignIn()
{
    TsgcHTTPComponentClient_OAuth2_Microsoft *oClient = new TsgcHTTPComponentClient_OAuth2_Microsoft.Create(this);
    Try
    {
        TsgcOAuth2_Microsoft_Data *oData = oClient.Authenticate("tenant_id", "client_id", "client_secret");
        if oData->Authenticated() then
            ShowMessage(oData->UserProfile->DisplayName);
    }
    __Finally
        oClient->Free();
}
```

OAuth2 | 授权码许可（RFC 6749）

概述

授权码授权是最常见的 OAuth2 流程。它专为 Web 应用程序、桌面应用程序和移动应用程序设计，其中客户端可以安全地存储客户端密钥。该流程涉及将用户重定向到授权服务器，在那里他们进行身份验证并授予权限。然后服务器将授权码返回到重定向 URI，客户端用该代码交换访问令牌。

流程

1. 客户端使用 `client_id`、`redirect_uri`、`response_type=code`、`scope` 和 `state` 参数将用户重定向到 **AuthURL**。
2. 用户向授权服务器进行身份验证并向客户端授予权限。
3. 授权服务器通过携带 `code` 参数的重定向 URI 将用户重定向回来。
4. 客户端将授权码与 `client_id`、`client_secret` 和 `redirect_uri` 一起 POST 到 **TokenURL**，以换取访问令牌。
5. 服务器返回 `access_token`、`token_type`、`expires_in`，以及可选的 `refresh_token`。

配置

属性	描述
<code>OAuth2Options.GrantType</code>	设置为 auth2Code 。
<code>OAuth2Options.ClientId</code>	授权服务器颁发的客户端标识符。
<code>OAuth2Options.ClientSecret</code>	授权服务器颁发的客户端密钥。
<code>AuthorizationServerOptions.AuthURL</code>	用户被重定向以进行身份验证的授权端点 URL。
<code>AuthorizationServerOptions.TokenURL</code>	用于将授权码交换为访问令牌的令牌端点 URL。
<code>AuthorizationServerOptions.Scope</code>	访问请求的范围（例如 <code>openid</code> 、 <code>profile</code> 、 <code>email</code> ）。
<code>LocalServerOptions.IP</code>	本地重定向服务器的 IP 地址（例如 <code>127.0.0.1</code> ）。
<code>LocalServerOptions.Port</code>	本地重定向服务器的端口（例如 <code>8080</code> ）。

事件

事件	描述
<code>OnBeforeAuthorizeCode</code>	在客户端将用户重定向到授权端点之前触发。允许自定义授权请求。
<code>OnAfterAuthorizeCode</code>	从服务器收到授权码后触发。
<code>OnErrorAuthorizeCode</code>	在授权码步骤出现错误时触发。
<code>OnBeforeAccessToken</code>	在客户端将授权码交换为访问令牌之前触发。
<code>OnAfterAccessToken</code>	收到访问令牌后触发。使用此事件检索令牌值。

OnErrorAccessToken

令牌交换步骤期间发生错误时触发。

示例

```
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();
```

令牌刷新

当访问令牌过期时，您可以使用刷新令牌在无需用户交互的情况下获取新令牌。使用刷新令牌值调用 `Refresh` 方法：`OnAfterAccessToken` 事件将使用新的访问令牌触发。如果刷新失败，`OnErrorAccessToken` 事件将触发。

何时使用

以下情况使用授权码授权：

- 具有可安全存储客户端密钥的服务器端后端的 Web 应用程序。
- 服务器端渲染应用程序（例如 ASP.NET、PHP、Java）。
- 可以合理保护客户端密钥的桌面应用程序。

对于无法安全存储客户端密钥的公共客户端（原生应用、SPA），请考虑改用[带 PKCE 的授权码流程](#)。

OAuth2 | 带 PKCE 的授权码（RFC 7636）

概述

带有 PKCE（Proof Key for Code Exchange）的授权码授权扩展了标准授权码流程，增加了一个额外的安全层。它专为原生应用程序、移动应用程序和单页应用程序（SPA）设计，因为这些应用程序无法安全存储客户端密钥。客户端不依赖客户端密钥，而是生成一个加密代码验证器和挑战值，以证明交换授权码的实体与发起流程的实体是同一个。

PKCE 工作原理

1. 客户端生成随机 `code_verifier`（32 字节密码学随机数据，Base64URL 编码）。
 2. 客户端将 `code_challenge` 计算为 `code_verifier` 的 SHA-256 哈希值，并进行 Base64URL 编码。
 3. 客户端在授权请求中发送 `code_challenge` 和 `code_challenge_method=S256`。
 4. 用户授权后，客户端在令牌交换请求中发送原始的 `code_verifier`。
 5. 授权服务器在颁发令牌之前验证 `SHA256(code_verifier) == code_challenge`。
- 这可以防止授权码拦截攻击，因为拦截了代码的攻击者在没有原始 `code_verifier` 的情况下无法交换它。

配置

属性	描述
<code>OAuth2Options.GrantType</code>	设置为 auth2CodePKCE 。
<code>OAuth2Options.ClientId</code>	授权服务器颁发的客户端标识符。
<code>OAuth2Options.ClientSecret</code>	可选。某些提供商即使使用 PKCE 也需要它；其他提供商则不需要。
<code>AuthorizationServerOptions.AuthURL</code>	用户被重定向以进行身份验证的授权端点 URL。
<code>AuthorizationServerOptions.TokenURL</code>	用于将授权码交换为访问令牌的令牌端点 URL。
<code>AuthorizationServerOptions.Scope</code>	访问请求的范围。
<code>LocalServerOptions.IP</code>	本地重定向服务器的 IP 地址（例如 127.0.0.1）。
<code>LocalServerOptions.Port</code>	本地重定向服务器的端口。设为 0 可使用随机可用端口。

示例

```
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();
```

随机端口

当 `LocalServerOptions.Port` 设置为 **0** 时，组件会自动选择一个随机可用端口。推荐桌面和移动应用程序使用此设置，因为它可以避免端口冲突。所选端口会包含在发送给授权服务器的重定向 URI 中。

令牌刷新

PKCE 流程通常返回刷新令牌。使用 `Refresh` 方法获取新的访问令牌：

何时使用

在以下情况使用带 PKCE 的授权码授权：

- 桌面应用程序（原生 Windows、macOS、Linux）。
- 移动应用程序（iOS、Android）。
- 在浏览器中运行的单页面应用程序（SPA）。
- 任何无法安全存储客户端密钥的公共客户端。
- 作为标准[授权码](#)授权的更安全替代方案，即使对于机密客户端也适用。

OAuth2 | 客户端凭证授权（RFC 6749）

概述

客户端凭证授权用于不需要用户交互的机器到机器（M2M）身份验证。客户端使用自己的凭证（`client_id` 和 `client_secret`）直接向授权服务器进行身份验证，并获得访问令牌。没有授权码步骤，也没有用户登录。此授权类型适用于客户端代表自身而非代表用户行动的场景。

流程

1. 客户端将 `client_id`、`client_secret`、`grant_type=client_credentials` 和 `scope` POST 到令牌端点。
2. 授权服务器验证客户端凭证。
3. 服务器返回 `access_token`、`token_type` 和 `expires_in`。

请注意，此授权类型通常不颁发刷新令牌，因为客户端可以直接使用其凭证请求新的访问令牌。

配置

属性	描述
<code>OAuth2Options.GrantType</code>	设为 auth2ClientCredentials 。
<code>OAuth2Options.ClientId</code>	授权服务器颁发的客户端标识符。
<code>OAuth2Options.ClientSecret</code>	授权服务器颁发的客户端密钥。
<code>AuthorizationServerOptions.TokenURL</code>	令牌端点 URL。由于没有用户授权步骤，无需 AuthURL。
<code>AuthorizationServerOptions.Scope</code>	访问请求的范围。

示例

```
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();
```

何时使用

在以下情况下使用客户端凭证许可：

- 无需用户交互即可运行的后端服务和守护进程。
- 服务器到服务器 API 通信。
- 计划任务、批处理进程和定时任务。
- 微服务间身份验证。
- 应用程序代表自身而非代表用户运行的任何场景。

OAuth2 | 资源所有者密码凭据授权（RFC 6749）

概述

资源所有者密码凭据（ROPC）授权允许客户端直接收集用户的用户名和密码，并将其换取访问令牌。用户凭据被发送到令牌端点，如果验证通过，服务器返回访问令牌。
仅当资源所有者与客户端之间存在高度信任时，才应使用此授权类型。

安全警告

此授权类型被认为比其他 OAuth2 流程安全性更低。它将用户凭据直接暴露给客户端应用程序，违反了委托授权原则。OAuth 2.1 规范已弃用此授权类型。请尽量改用[带 PKCE 的授权码](#)。

流程

1. 用户直接向客户端应用程序提供用户名和密码。
2. 客户端将 username、password、client_id、client_secret、grant_type=password 和 scope 以 POST 方式提交到令牌端点。
3. 授权服务器验证凭据。
4. 服务器返回 access_token、token_type、expires_in，以及可选的 refresh_token。

配置

属性	描述
<code>OAuth2Options.GrantType</code>	设为 <code>auth2ResourceOwnerPassword</code> 。
<code>OAuth2Options.ClientId</code>	授权服务器颁发的客户端标识符。
<code>OAuth2Options.ClientSecret</code>	授权服务器颁发的客户端密钥。
<code>OAuth2Options.UserName</code>	资源所有者的用户名。
<code>OAuth2Options.Password</code>	资源所有者的密码。
<code>AuthorizationServerOptions.TokenURL</code>	令牌端点 URL。由于用户凭据直接提供，不需要 AuthURL。
<code>AuthorizationServerOptions.Scope</code>	访问请求的范围。

示例

```
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();
```


令牌刷新

如果服务器返回刷新令牌，您可以使用它在不重新输入用户凭据的情况下获取新的访问令牌：

何时使用

仅在以下情况下使用资源所有者密码凭据授权：

- 无法迁移到更安全流程的旧版应用程序。
- 您同时控制客户端和授权服务器的第一方应用程序。
- 正在从直接用户名/密码认证过渡到 OAuth2 的迁移场景。

对于所有其他情况，请改用 [带 PKCE 的授权码](#) 或 [授权码](#)。

OAuth2 | 设备授权授权（RFC 8628）

概述

设备授权许可（也称为设备代码流）专为无法轻松显示浏览器或接受键盘输入的受限设备而设计。示例包括智能电视、IoT 设备、媒体播放器、CLI 工具和信息亭。设备显示一个短代码，用户在另一台设备（如手机或电脑）上输入该代码以完成授权。

流程

1. 客户端将 `client_id` 和 `scope` POST 到 **DeviceAuthorizationURL**。
2. 服务器返回 `device_code`、`user_code`、`verification_uri`、`expires_in` 和 `interval`。
3. 客户端触发 **OnDeviceCode** 事件，提供用户代码和验证 URI。应用程序将这些显示给用户（例如："请访问 `https://provider.com/device` 并输入代码：ABCD-1234"）。
4. 用户在另一设备上导航至验证 URI 并输入用户代码。
5. 用户在辅助设备上完成身份验证并授权。
6. 同时，客户端每隔 `interval` 秒使用设备码自动轮询令牌端点。
7. 用户完成授权后，下一次轮询将返回 `access_token`。
8. 如果在用户授权之前设备代码过期，将触发 **OnDeviceCodeExpired** 事件。

配置

属性	描述
<code>OAuth2Options.GrantType</code>	设置为 auth2DeviceCode 。
<code>OAuth2Options.ClientId</code>	授权服务器颁发的客户端标识符。
<code>OAuth2Options.ClientSecret</code>	可选。某些提供商要求，其他不要求。
<code>AuthorizationServerOptions.TokenURL</code>	用于轮询的令牌端点 URL。
<code>AuthorizationServerOptions.DeviceAuthorizationURL</code>	客户端请求设备代码的设备授权端点。
<code>AuthorizationServerOptions.Scope</code>	访问请求的范围。

事件

事件	描述
<code>OnDeviceCode</code>	当从服务器收到设备代码和用户代码时触发。向用户显示用户代码和验证 URI。
<code>OnDeviceCodeExpired</code>	当设备码在用户完成授权之前过期时触发。应用程序应提示用户重新开始流程。
<code>OnAfterAccessToken</code>	当用户完成授权且收到访问令牌时触发。
<code>OnErrorAccessToken</code>	当设备流程中发生不可恢复的错误时触发。

示例

处理设备代码事件

```
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();
```

轮询行为

调用 `Start` 后，组件会按服务器指定的间隔（通常为 5 秒）自动轮询令牌端点。如果服务器返回 `slow_down` 错误，组件会将轮询间隔增加 5 秒。轮询持续进行，直到发生以下任一情况：

- 用户完成授权后，令牌被返回。
- 设备码已过期（触发 `OnDeviceCodeExpired`）。
- 发生不可恢复的错误（触发 `OnErrorAccessToken`）。

何时使用

在以下情况使用设备授权授权：

- 智能电视和流媒体设备。
- 命令行界面（CLI）工具。
- 输入能力有限的 IoT 设备。
- 信息亭和数字标牌。
- 任何输入完整 URL 或凭据不便的设备。

OAuth2 | DPOP — 展示持有证明 (RFC 9449)

概述

DPoP (持有证明) 是一种安全机制, 通过要求证明持有私钥, 将访问令牌绑定到特定客户端。与任何获取者都可使用的 Bearer 令牌不同, DPoP 绑定的令牌在没有对应私钥的情况下毫无用处。这为令牌窃取和重放攻击提供了防护。

DPoP 本身不是一种授权类型, 而是一种安全增强措施, 可与任何 OAuth2 授权类型 (授权码、客户端凭据等) 一起使用。

工作原理

1. 客户端生成一个非对称密钥对 (ES256 或 RS256)。
2. 客户端创建具有以下结构的 **DPoP 证明 JWT** :
 - 标头: typ: dpop+jwt、alg: ES256、jwk: {public key}
 - 载荷: htm (HTTP 方法)、htu (HTTP URI)、iat (签发时间)、jti (唯一标识符)
3. 客户端将 DPoP 证明作为 HTTP 标头 (DPoP: <proof>) 与令牌请求一起发送。
4. 授权服务器将已颁发的令牌绑定到客户端公钥的 JWK 指纹。
5. 服务器返回 token_type: DPoP 而非 token_type: Bearer。
6. 对于后续 API 调用, 客户端同时发送 Authorization: DPoP <token> 头和特定于该请求的新 DPoP: <proof> 头。

密钥生成

您可以使用 **GenerateDPoPKeyPair** 方法直接在 Delphi 中生成 ES256 密钥对。此方法在内部使用 OpenSSL 创建密钥对, 并自动设置 PrivateKey 和 PublicKeyJWK :

```
OAuth2->DPoPOptions->Enabled = true;
OAuth2->DPoPOptions->Algorithm = dpopES256;
OAuth2->GenerateDPoPKeyPair(); // Generates PrivateKey + PublicKeyJWK automatically
OAuth2->Start();
```

PublicKeyToJWK — 将任意密钥或证书转换为 JWK

如果您已有公钥、私钥或 X.509 证书 (PEM 格式), 可以使用 **PublicKeyToJWK** 类函数将其转换为 JWK。它接受以上任意输入并返回正确的 JWK JSON :

输入	PEM 头
EC 或 RSA 公钥	-----BEGIN PUBLIC KEY-----
EC 私钥	-----BEGIN EC PRIVATE KEY-----
RSA 私钥	-----BEGIN RSA PRIVATE KEY-----
X.509 证书	-----BEGIN CERTIFICATE-----

从 PEM 密钥文件

```
// 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"));
```

来自智能卡证书

智能卡和 HSM 保护私钥，私钥无法被读取或导出。然而，证书（包含公钥）始终可以导出。将证书导出为 PEM 格式并传递给 `PublicKeyToJWK`：

```
// Export the certificate from the smart card
UnicodeString CertPEM = MySmartCard->ExportCertificatePEM();

// Convert the certificate to JWK
OAuth2->DPoPOptions->PublicKeyJWK =
    OAuth2->PublicKeyToJWK(CertPEM);

// No PrivateKey needed - use OnDPoPSign for signing via the card
OAuth2->DPoPOptions->Enabled = true;
OAuth2->OnDPoPSign = MyDPoPSignHandler;
```

此方法适用于任何可以导出其证书的智能卡或 HSM：PKCS#11 令牌、Windows CNG 智能卡、USB 安全密钥（YubiKey、Feitian）以及云 HSM（Azure Key Vault、AWS CloudHSM）。

使用 OpenSSL 命令行工具

您也可以在外部分生成密钥对：

DPoP 选项

属性	描述
<code>DPoPOptions.Enabled</code>	设置为 True 可启用 DPoP 证明生成。
<code>DPoPOptions.Algorithm</code>	签名算法。支持的值： ES256 （推荐）、 RS256 。
<code>DPoPOptions.PrivateKey</code>	用于签署 DPoP 证明 JWT 的 PEM 编码私钥。
<code>DPoPOptions.PublicKeyJWK</code>	DPoP 证明头部中包含的 JWK（JSON Web Key）格式公钥。

方法

方法	描述
<code>GenerateDPoPKeyPair</code>	使用 OpenSSL 生成 ES256 密钥对，并自动设置 <code>DPoPOptions.PrivateKey</code> （PEM）和 <code>DPoPOptions.PublicKeyJWK</code> （JSON）。
<code>GetDPoPProof(Method, URL, AccessToken)</code>	为给定的 HTTP 方法、URL 和访问令牌生成 DPoP 证明 JWT，并返回已签名的 JWT 字符串。该证明包含 <code>ath</code> 声明（访问令牌哈希），以将证明绑定到令牌。
<code>GetDPoPJWKThumbprint</code>	返回公钥的 RFC 7638 JWK 指纹（SHA-256），服务器用其绑定令牌。

Nonce 处理

某些授权服务器要求在 DPoP 证明中包含服务器提供的 nonce。当服务器以 400 或 401 状态响应并附带 DPoP-Nonce 响应头时，组件将自动：

1. 从 DPoP-Nonce 响应标头中提取 nonce。

2. 在下一个 DPOP 证明 JWT 的 nonce 声明中包含 nonce。
3. 使用更新后的证明重试请求。

使用 OnDPoPSign 的自定义签名（HSM / 智能卡）

在某些环境中，私钥存储在硬件安全模块（HSM）或智能卡上（例如通过 PKCS#11 或 Windows CNG），无法提取。OnDPoPSign 事件允许您在外部执行签名，而组件处理其他所有事项（证明结构、nonce、请求头）。当分配了 OnDPoPSign 时，组件会在尝试内部签名之前调用它。如果您将 `Handled := True` 并提供 `Signature`，组件将使用您的签名并跳过内部签名。此情况下不需要 `DPoPOptions.PrivateKey` 属性。

事件签名

参数	方向	描述
<code>SigningInput</code>	in	待签名数据： <code>Base64URL(header) + "." + Base64URL(payload)</code> 。这是标准的 JWS 签名输入。
<code>Algorithm</code>	in	算法名称： <code>"ES256"</code> 或 <code>"RS256"</code> 。
<code>Signature</code>	out	将此设置为您的签名设备生成的 Base64URL 编码签名字节。
<code>Handled</code>	out	设置为 <code>True</code> 以使用您的自定义签名。如果为 <code>False</code> ，组件将回退到使用 <code>PrivateKey</code> 进行内部签名。

智能卡示例

以下示例展示了如何将 DPOP 与签名卡一起使用。私钥从不离开卡，只有哈希值被发送到卡进行签名。

```
// Configure DPOP - no PrivateKey needed when using OnDPoPSign
OAuth2->DPoPOptions->Enabled = true;
OAuth2->DPoPOptions->Algorithm = dpopES256;
OAuth2->DPoPOptions->PublicKeyJWK =
    "{\"kty\":\"EC\",\"crv\":\"P-256\",\"x\":\"...\",\"y\":\"...\"}";
OAuth2->OnDPoPSign = OnDPoPSignHandler;
OAuth2->Start();

void __fastcall TForm1::OnDPoPSignHandler(TObject *Sender,
    const UnicodeString SigningInput, const UnicodeString Algorithm,
    UnicodeString &Signature, bool &Handled)
{
    // Hash and sign with smart card
    TBytes Hash = GetHashSHA256(SigningInput);
    TBytes Sig = MySmartCardAPI->SignHash(Hash);
    Signature = EncodeBase64URL(Sig);
    Handled = true;
}
```

此方法适用于任何签名后端：PKCS#11 令牌、Windows CNG（CryptoAPI Next Generation）、Azure Key Vault、AWS CloudHSM，或任何能够生成 ECDSA/RSA 签名的自定义硬件。

示例

```
// Configure the grant type
OAuth2->OAuth2Options->GrantType = auth2CodePKCE;
OAuth2->OAuth2Options->ClientId = "your-client-id";
OAuth2->AuthorizationServerOptions->AuthURL = "https://provider.com/oauth2/authorize";
OAuth2->AuthorizationServerOptions->TokenURL = "https://provider.com/oauth2/token";
OAuth2->AuthorizationServerOptions->Scope->Text = "openid profile";
// Enable DPOP
OAuth2->DPoPOptions->Enabled = true;
OAuth2->DPoPOptions->Algorithm = "ES256";
OAuth2->DPoPOptions->PrivateKey = LoadPrivateKeyFromFile("dpop_private.pem");
OAuth2->DPoPOptions->PublicKeyJWK = "{\"kty\":\"EC\",\"crv\":\"P-256\",\"x\":\"...\",\"y\":\"...\"}";
OAuth2->Start();
```

```
// Generate DPoP proof for API calls  
UnicodeString Proof = OAuth2->GetDPoPProof("GET", "https://api.provider.com/resource");
```

TsgcHTTP_OAuth2_Server

此组件在服务器端组件中提供 OAuth2 协议实现。

简介

此组件在服务器端组件中提供 OAuth2 协议实现。

服务器组件有一个名为 Authorization.OAuth.OAuth2 的属性，您可以在其中分配一个 TsgcHTTP_OAuth2_Server 实例，因此如果启用了 Authentication 且 OAuth2 属性附加了 OAuth2 服务器组件，则 WebSocket 和 HTTP 请求需要 Bearer 令牌才能被处理，否则连接将自动关闭。

```
TsgcHTTP_OAuth2_Server *OAuth2 = new TsgcHTTP_OAuth2_Server(this);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2;
```

服务器支持以下授权类型：

- **auth2Code**：用于在大多数应用程序类型（包括单页应用、Web 应用和原生安装应用）中执行身份验证和授权。该流程使应用能够安全获取可用于访问受保护资源的 access_token，以及用于获取额外 access_token 的刷新令牌，以及已登录用户的 ID 令牌。
- **auth2ClientCredentials**：此授权类型通常用于必须在后台运行且无需与用户即时交互的服务器间交互。此类应用程序通常称为守护进程或服务账户。
- **password**（资源所有者密码凭据）：允许应用程序通过直接处理用户凭据来登录用户。客户端将用户的用户名和密码发送到令牌端点。
- **urn:ietf:params:oauth:grant-type:device_code**（设备码）：根据 RFC 8628 的设备授权许可。允许输入受限的设备（智能电视、IoT 设备）通过让用户在辅助设备上授权的方式获取用户授权。

注册应用程序时可以自定义授权类型，默认情况下支持所有授权类型。

端点

默认情况下，组件配置以下端点来处理授权和令牌请求

Authorization：/sgc/oauth2/auth

Token：/sgc/oauth2/token

Revocation：/sgc/oauth2/revoke

自省：/sgc/oauth2/introspect

设备授权：/sgc/oauth2/device

设备验证：/sgc/oauth2/device/verify

因此，如果服务器在端口 443 上监听且域名为 www.esegece.com，则端点为：

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

设备验证：https://www.esegece.com/sgc/oauth2/device/verify

端点可在 OAuth2Options 属性中配置。

默认情况下，启用了 PKCE（这是授权码流程的扩展，用于防止 CSRF 和授权码注入攻击）。

配置

在开始 OAuth2 流程之前，您必须注册将要使用的应用程序，这可通过 OAuth2 服务器组件的 Apps 属性完成。

注册应用程序

使用 Apps.AddApp 向 OAuth2 服务器添加新应用程序，必须设置以下参数：

- **应用名称**：是应用程序的名称，示例：MyApp
- **RedirectURI**：响应将被重定向到的地址。示例：http://127.0.0.1:8080
- **ClientId**：这是公开信息，即客户端的 ID。

- **ClientSecret**：必须保密。

您可以选择设置以下参数：

- **ExpiresIn**：默认为 3600 秒，即令牌将在 1 小时后过期，若有需要可设置更大的值。
- **RefreshToken**：默认情况下支持刷新令牌，如不支持，请将此参数设置为 false。
- **AllowedGrantTypes**：默认情况下支持所有授权类型（auth2Code 和 auth2ClientCredentials），但服务器可配置为仅允许授权码模式或仅允许客户端凭证模式。

删除应用

使用 `Apps.RemoveApp` 删除现有应用程序。

AddToken

如果服务器在颁发了一些令牌后重新启动，您可以在启动 OAuth2 服务器并注册应用程序之前，使用 `AddToken` 方法恢复这些令牌。

- **AppName**: 应用程序的名称。
- **Token**：访问令牌。
- **过期时间**：令牌的过期时间。
- **RefreshToken**：刷新令牌。

RemoveToken

删除已颁发的令牌。

OAuth2Options

`OAuth2Options` 属性允许配置服务器端点和可选功能。

吊销

符合 [RFC 7009](#) 的令牌撤销。启用后，客户端可撤销之前颁发的访问令牌或刷新令牌。

- **OAuth2Options.Revocation.Enabled**：设置为 True 以后启用撤销端点。
- **OAuth2Options.Revocation.URL**：端点 URL 路径。默认值：`/sgc/oauth2/revoke`

自省

按照 [RFC 7662](#) 进行令牌自省。启用后，资源服务器可查询授权服务器以确定令牌的活动状态和元数据。

- **OAuth2Options.Introspection.Enabled**：设置为 True 以后用自省端点。
- **OAuth2Options.Introspection.URL**：端点 URL 路径。默认值：`/sgc/oauth2/introspect`

DeviceAuthorization

设备授权授予方式，符合 [RFC 8628](#)。启用后，输入受限的设备可通过让用户在辅助设备上授权来请求授权。

- **OAuth2Options.DeviceAuthorization.Enabled**：设为 True 以后用设备授权端点。
- **OAuth2Options.DeviceAuthorization.URL**：设备代码请求的端点 URL 路径。默认值：`/sgc/oauth2/device`
- **OAuth2Options.DeviceAuthorization.VerificationURL**：用户验证页面的端点 URL 路径。默认：`/sgc/oauth2/device/verify`
- **OAuth2Options.DeviceAuthorization.ExpiresIn**：设备码的生命周期（秒）。默认值：600（10 分钟）。
- **OAuth2Options.DeviceAuthorization.Interval**：客户端轮询令牌端点时应使用的最小轮询间隔（以秒为单位）。默认值：5。

参考

- 属性——组件上提供 2 个属性。
- 方法 — 11 个公共方法。

- 事件——13 个可处理的事件。

指南

- 快速入门
 - [OAuth2 服务器示例](#)
 - [OAuth2 自定义登录 HTML](#)
 - [OAuth2 服务器端点](#)
 - [OAuth2 注册应用程序](#)
 - [OAuth2 恢复访问令牌](#)
- 认证
 - [OAuth2 服务器认证](#)
 - [OAuth2 无需认证的 URL](#)

OAuth2 | 服务器示例

让我们使用 [TsgcWebSocketHTTPServer](#) 做一个简单的 OAuth2 服务器示例。

首先，创建一个新的 [TsgcWebSocketHTTPServer](#)，监听 443 端口，并使用 `sgc.pem` 文件中的自签名证书。

```
oServer = new TsgcWebSocketHTTPServer(this);
oServer->Port := 80;
oServer->SSLOptions->Port = 443;
oServer->SSLOptions->CertFile = "sgc.pem";
oServer->SSLOptions->KeyFile = "sgc.pem";
oServer->SSLOptions->RootCertFile = "sgc.pem";
oServer->SSL = true;
```

然后创建 [TsgcHTTP_OAuth2_Server](#) 的新实例，并将其分配给之前创建的服务器。
使用以下值注册新应用程序：

Name: MyApp
RedirectURL: <http://127.0.0.1:8080>
ClientId: client-id
ClientSecret: client-secret

```
OAuth2 = new TsgcHTTP_OAuth2_Server.Create(this);
OAuth2->Apps->AddApp("MyApp", "http://127.0.0.1:8080", "client-id", "client-secret");
oServer->Authentication->Enabled = true;
oServer->Authentication->OAuth->OAuth2 = OAuth2;
```

然后处理 OAuth2 服务器组件的 `OnOAuth2Authentication` 事件，并实现您自己的用户登录方法。此处将使用 "user/secret" 这一组合来接受登录。

```
void OnOAuth2Authentication(TsgcWSConnection *Connection, TsgcHTTPOAuth2Request *OAuth2, string aUser,
    string aPassword, ref bool Authenticated)
{
    if ((aUser == "user") and (aPassword == "secret"))
    {
        Authenticated = true;
    }
}
```

最后，启动服务器并使用 OAuth2 客户端登录，例如可以使用 [sgcWebSockets](#) 库中包含的 [TsgcHTTP_OAuth2_Client](#)。

OAuth2

Configuration

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

Port

8080

Redirect URL

OAuth2 Options

ClientId

client-id

Secret

client-secret

Username

Password

New Access Token

Access Token

Token Type

Expires In

Refresh Token

Refresh Token

Scope

请求新的访问令牌，将显示一个新的 Web 浏览器会话，用户必须允许连接然后登录。

OAuth2 Authorization

Sign in

Introduce your username and password.

GO BACK

Sign in

SIGN IN

如果登录成功，将向客户端返回新令牌。之后所有请求必须在 HTTP 头中包含此 Bearer 令牌。

 OAuth2

Configuration

Gmail

Auth. URL

https://127.0.0.1/sgc/oauth2/auth

Token URL

https://127.0.0.1/sgc/oauth2/token

Scope

scope

Authorization Server Options

IP

127.0.0.1

Port

8080

Redirect URL

Local Server Options

ClientId

client-id

Secret

client-secret

Username

Password

OAuth2 Options

New Access Token

Access Token

be4d115a9e6a44f0a859cb303d7f03890d3bf290fc4342c1a08befa550b738bd

Token Type

Bearer

Expires In

3600

Refresh Token

b5ec418745ef4c9e94d3b1a90b34324826152b7edbf040a6a55271813aac5849

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 | 自定义登录 HTML

当 OAuth2 客户端请求获取新的访问令牌时，Web 浏览器中会显示一个网页，允许此连接并使用用户名和密码登录。

HTML 页面默认包含在 Server 组件中，但可以使用 **OnAuth2BeforeDispatchPage** 事件进行自定义。

```
void OnAuth2BeforeDispatchPage(TObject *Sender; TsgcHTTPOAuth2Request *OAuth2; ref string HTML)
{
    HTML = "your custom html";
}
```

如果您使用全新的 HTML 代码自定义 HTML，至少必须保留发送用户名和密码的表单：

```
<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>
```

隐藏的 id 参数必须保持与原始表单相同的值，以便服务器识别该请求。

OAuth2 | 服务器端点

默认情况下，OAuth2 服务器使用以下端点：

Authorization：/sgc/oauth2/auth

Token：/sgc/oauth2/token

撤销：/sgc/oauth2/revoke (POST) — 依据 [RFC 7009](#) 撤销令牌。

Introspection：/sgc/oauth2/introspect (POST) — 根据 [RFC 7662](#) 返回令牌元数据。

设备授权：/sgc/oauth2/device (POST) - 按照 [RFC 8628](#) 颁发设备代码

设备验证：/sgc/oauth2/device/verify (GET/POST) - 用户验证页面

这意味着，如果您的服务器监听 IP 80.54.41.30、端口 8443，则完整的 OAuth2 端点将为：

Authorization：https://80.54.41.30:8443/sgc/oauth2/auth

Token：https://80.54.41.30:8443/sgc/oauth2/token

吊销：https://80.54.41.30:8443/sgc/oauth2/revoke

自省：https://80.54.41.30:8443/sgc/oauth2/introspect

设备授权：https://80.54.41.30:8443/sgc/oauth2/device

设备验证：https://80.54.41.30:8443/sgc/oauth2/device/verify

这些端点可以轻松修改，只需访问组件的 OAuth2Options 属性并修改授权和令牌 URL 即可。

示例：如果您的端点必须是

Authorization：https://80.54.41.30:8443/authentication/auth

Token：https://80.54.41.30:8443/authentication/token

使用以下值设置 OAuth2Options 属性：

OAuth2Options.Authorization.URL = /authentication/auth

OAuth2Options.Token.URL = /authentication/token

同样的方法适用于其他端点：

OAuth2Options.Revocation.URL = /authentication/revoke

OAuth2Options.Introspection.URL = /authentication/introspect

OAuth2Options.DeviceAuthorization.URL = /authentication/device

OAuth2Options.DeviceAuthorization.VerificationURL = /authentication/device/verify

OAuth2 | Register Apps

在客户端请求新的 OAuth2 之前，应用程序必须在服务器中注册。
注册新应用程序需要以下信息：

- **应用名称**：是应用程序的名称，示例：MyApp
- **RedirectURI**：响应将被重定向到的地址。示例：http://127.0.0.1:8080
- **ClientId**：这是公开信息，即客户端的 ID。
- **ClientSecret**：必须保密。

您可以选择设置以下参数：

- **ExpiresIn**：默认为 3600 秒，即令牌将在 1 小时后过期，若有需要可设置更大的值。
- **RefreshToken**：默认情况下支持刷新令牌，如不支持，请将此参数设置为 false。

若新客户端希望使用 OAuth2 进行认证，首先需要在服务器中注册应用程序，您可以使用：

1. RegisterApp

此方法需要应用名称和 RedirectURI，并将返回 ClientId 和 ClientSecret。

2. Apps.AddApp

此方法需要 AppName、RedirectURI、ClientId 和 ClientSecret。通常，当服务器已创建了一些应用并希望在启动前加载它们时，可以使用此方法。

两种方法的作用相同，都是在服务器中注册应用程序，但第一种方法在首次注册应用时更为常用，第二种方法适用于在启动服务器前加载所有已注册应用的场景（例如已保存在数据库中）。

OAuth2 | 恢复访问令牌

若 OAuth2 服务器被销毁（例如因重启），已颁发的有效访问令牌将默认丢失。您可以使用方法 **AddToken** 恢复这些访问令牌。此方法将令牌存储在 OAuth2 服务器中。

添加令牌需要以下信息：

- **AppName**：应用程序的名称。
- **Token**：访问令牌。
- **过期时间**：令牌的过期时间。
- **RefreshToken**：刷新令牌。

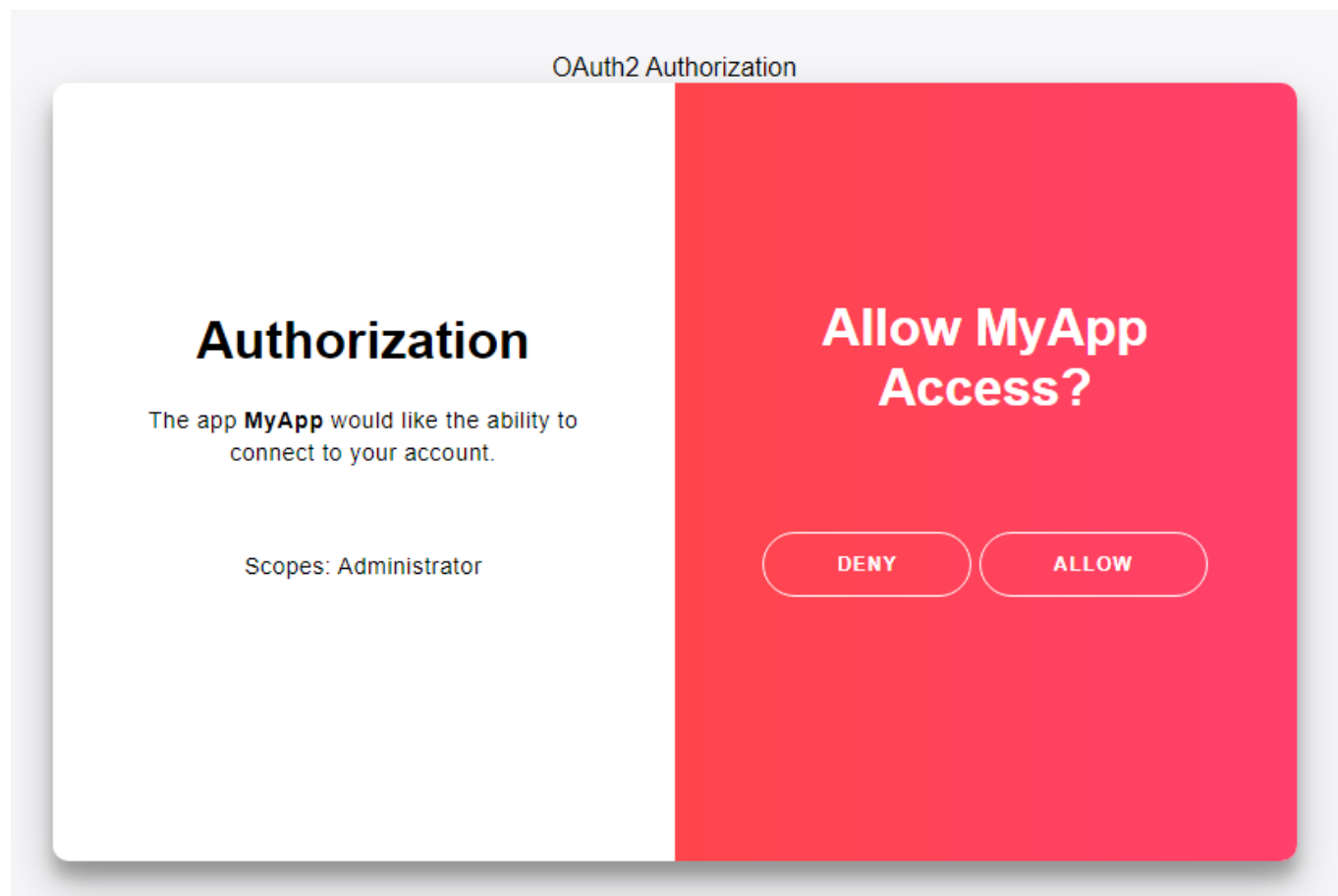
您可以通过处理 **OAuth2AfterAccessToken** 事件来保存已签发的令牌。

```
private void OnOAuth2AfterAccessToken(TObject *Sender, TsgcWConnection *Connection, TsgcHTTPOAuth2Request *OAuth2Request, TsgcHTTPOAuth2Response *aResponse)
{
    // ... store OAuth2 Token data
}

OAuth2 = new TsgcHTTP_OAuth2_Server.Create(this);
OAuth2->Apps->AddApp("MyApp", "http://127.0.0.1:8080", "client-id", "client-secret");
OAuth2->AddToken("MyApp", "12146ce12b0e4813987f2794f768905cefc39da6fbd54f6d9b38387489280608", EncodeDate(2022, 1, 1), "ef3e3dfa56ec44109c3d345b1541f08e539ce21432d9433099b48a3d08d34bc0");
oServer->Authentication->Enabled = true;
oServer->Authentication->OAuth->OAuth2 = OAuth2;
```

OAuth2 | 服务器认证

当 OAuth2 客户端请求新授权时，服务器会显示一个网页，用户必须在该页面上允许连接然后登录。此页面由 sgcWebSockets 库提供，并在客户端请求授权时自动分发。



如果用户允许访问，将显示登录表单，用户须输入用户名和密码。此数据将在 OnOAuth2Authentication 事件中接收，因此您必须验证用户名/密码是否正确，若正确则将 Authenticated 参数设为 true。

```
void OnOAuth2Authentication(TsgcWSConnection *Connection, TsgcHTTPOAuth2Request *OAuth2, string aUser,
    string aPassword, ref bool Authenticated)
{
    if ((aUser == "user") and (aPassword == "secret"))
    {
        Authenticated = true;
    }
}
```

OAuth2 | 无认证 URL

默认情况下，当服务器端启用 OAuth2 时，所有 HTTP 请求都需要使用 Bearer Token 进行身份验证。
若要允许某些 URL 无需使用 Bearer Token 即可访问，可使用事件 **OnOAuth2BeforeRequest**

```
procedure OnOAuth2BeforeRequest(TObject *Sender; TsgcWSConnection *aConnection; TStringList *aHeaders;  
    ref bool Cancel)  
{  
    if (DecodeGETFullPath(aHeaders) == "/Public.html")  
    {  
        Cancel = true;  
    }  
}
```

OAuth2 服务器 | 授权码授权

概述

授权码授权是服务器端应用程序最常见的 OAuth2 流程。[TsgcHTTP_OAuth2_Server](#) 提供两个端点来处理此流程：一个向用户显示登录页面的授权端点，以及一个将授权码换取访问令牌的令牌端点。此流程适用于客户端能够与浏览器交互的 Web 应用程序、桌面应用程序和移动应用程序。

流程

1. 客户端将用户重定向到 `/sgc/oauth2/auth?`
`response_type=code&client_id=...&redirect_uri=...&scope=...&state=...`
2. 服务器触发 **OnOAuth2BeforeDispatchPage** 以允许自定义 HTML 登录页面。
3. 用户提交凭证。服务器触发 **OnOAuth2Authentication** 以验证用户名和密码。
4. 如果验证通过，服务器会将用户重定向到 `redirect_uri`，并附加 `?code=...&state=...`。
5. 客户端向 `/sgc/oauth2/token` 发送 POST，参数为
`grant_type=authorization_code&code=...&redirect_uri=...&client_id=...&client_secret=...`
6. 服务器验证授权码和客户端凭据，触发 **OnOAuth2AfterAccessToken**，并返回包含 `access_token`、`token_type`、`expires_in` 以及可选 `refresh_token` 的 JSON 响应。

配置

属性	描述
<code>OAuth2Options.Authorization.Enabled</code>	设置为 True 以后用授权端点。默认值：True。
<code>OAuth2Options.Authorization.URL</code>	授权端点 URL 路径。默认值： <code>/sgc/oauth2/auth</code> 。
<code>OAuth2Options.Token.Enabled</code>	设为 True 以后用令牌端点。默认值：True。
<code>OAuth2Options.Token.URL</code>	令牌端点 URL 路径。默认值： <code>/sgc/oauth2/token</code> 。
<code>OAuth2Options.PKCE</code>	为 True 时，服务器要求授权码请求必须提供 PKCE（ <code>code_challenge</code> 和 <code>code_verifier</code> ）。默认值：True。

应用注册

在客户端可以使用授权码流程之前，您必须在服务器上注册应用程序。使用 `Apps.AddApp` 并确保 `AllowedGrantTypes` 包含 `auth2Code`。

```
OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
    "my-client-id", "my-client-secret", 3600, true,
    TsgcOAuth2GrantTypes() << auth2Code);
```

事件

事件	描述
<code>OnOAuth2BeforeDispatchPage</code>	在将登录 HTML 页面发送给用户之前触发。允许自定义 HTML 内容。

<code>OnOAuth2Authentication</code>	当用户提交凭据时触发。在此验证用户名和密码，并将 <code>Authenticated</code> 设为 True 或 False。
<code>OnOAuth2AfterAccessToken</code>	服务器成功颁发访问令牌后触发，可用于日志记录。
<code>OnOAuth2AfterValidateAccessToken</code>	当客户端使用有效且未过期的令牌发起请求时触发。适用于日志记录和审计。

示例

以下示例展示了一个完整的服务器设置，使用授权码授权方式，包括应用注册和事件处理程序。

```
// Create and configure OAuth2 server
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);

// Register an application
OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
    "my-client-id", "my-client-secret", 3600, true,
    TsgcOAuth2GrantTypes() << auth2Code);

// Attach to HTTP server
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;

void __fastcall TForm1::OAuth2ServerOAuth2Authentication(TObject *Sender,
    const UnicodeString User, const UnicodeString Password, bool &Authenticated)
{
    Authenticated = (User == "admin") && (Password == "secret");
}

void __fastcall TForm1::OAuth2ServerOAuth2AfterAccessToken(TObject *Sender,
    const UnicodeString Token, const UnicodeString RefreshToken, int ExpiresIn)
{
    Log("Access token issued: " + Token);
}
```

PKCE 支持

当 `OAuth2Options.PKCE` 设置为 `True`（默认值）时，服务器强制执行 Proof Key for Code Exchange。客户端必须在授权请求中包含 `code_challenge` 参数，并在令牌请求中包含相应的 `code_verifier`。服务器在签发令牌前验证 `code_verifier` 与原始 `code_challenge` 匹配。

PKCE 推荐用于所有客户端，尤其是无法安全存储客户端密钥的公开客户端（原生应用和 SPA）。

OAuth2 服务器 | 客户端凭据授予

概述

Client Credentials 授权类型用于机器对机器 (M2M) 通信，无需用户交互。客户端使用自身凭证 (client_id 和 client_secret) 直接向 [TsgcHTTP_OAuth2_Server](#) 进行身份验证并获取访问令牌。没有授权码步骤、重定向或用户登录页面。

此授权类型适用于客户端应用程序代表自身而非特定用户进行操作的场景。

流程

1. 客户端使用 `grant_type=client_credentials&client_id=...&client_secret=...&scope=...` POST 到令牌端点
2. 服务器根据已注册的应用程序验证客户端凭据。
3. 服务器触发 **OnOAuth2AfterAccessToken** 并返回包含 `access_token`、`token_type` 和 `expires_in` 的 JSON 响应。

此授权类型不会触发 `OnOAuth2Authentication` 事件，因为没有需要认证的用户。服务器会自动根据已注册的应用验证 `client_id` 和 `client_secret`。

配置

使用 `AllowedGrantTypes` (包含 `auth2ClientCredentials`) 注册应用程序。此流程只需要令牌端点。

属性	描述
<code>OAuth2Options.Token.Enabled</code>	设为 True 以后用令牌端点。默认值：True。
<code>OAuth2Options.Token.URL</code>	令牌端点 URL 路径。默认值： <code>/sgc/oauth2/token</code> 。

应用注册

```
OAuth2Server->Apps->AddApp("MyServiceApp", "",
    "service-client-id", "service-client-secret", 3600, false,
    TsgcOAuth2GrantTypes() << auth2ClientCredentials);
```

`RedirectURI` 可以留空，因为不涉及重定向。`RefreshToken` 通常设置为 `False`，因为客户端可以随时使用其凭据请求新令牌。

事件

事件	描述
<code>OnOAuth2AfterAccessToken</code>	服务器成功颁发访问令牌后触发，可用于日志记录。
<code>OnOAuth2AfterValidateAccessToken</code>	当客户端使用有效令牌发出请求时触发。适用于审计。

示例

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->Apps->AddApp("BackendService", "",
    "service-client-id", "service-client-secret", 7200, false,
    TsgcOAuth2GrantTypes() << auth2ClientCredentials);
```

```
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;
void __fastcall TForm1::OAuth2ServerOAuth2AfterAccessToken(TObject *Sender,
    const UnicodeString Token, const UnicodeString RefreshToken, int ExpiresIn)
{
    Log("Service token issued: " + Token);
}
```

何时使用

在服务器端对以下情况使用客户端凭证授权：

- 无需用户交互即可进行身份验证的后端服务和守护进程。
- 服务器到服务器 API 通信。
- 微服务间身份验证。
- 自动化流程、计划任务和批量操作。

OAuth2 服务器 | 资源所有者密码凭据许可

概述

资源所有者密码凭据 (ROPC) 授权允许客户端直接将用户的用户名和密码发送到令牌端点。[TsgcHTTP_OAuth2_Server](#) 通过 **OnOAuth2Authentication** 事件验证凭据，并在身份验证成功时签发访问令牌。此流程不涉及浏览器重定向或登录页面。客户端直接从用户收集凭据并提交给服务器。

安全警告

此授权类型在 **OAuth 2.1** 中已弃用。仅当其他流程（如带 PKCE 的授权码）不可行时才应使用，例如在旧版应用程序或高度受信任的第一方客户端中。带 PKCE 的授权码授权是推荐的替代方案。

流程

1. 客户端以 `grant_type=password&username=...&password=...&client_id=...&client_secret=...&scope=...` 向令牌端点发起 POST 请求
2. 服务器触发 **OnOAuth2Authentication** 以验证用户名和密码。
3. 若认证成功，服务器触发 **OnOAuth2AfterAccessToken** 并返回包含 `access_token`、`token_type`、`expires_in` 以及可选 `refresh_token` 的 JSON 响应。
4. 若身份验证失败，服务器返回错误响应。

配置

使用包含 `auth2ResourceOwnerPassword` 的 `AllowedGrantTypes` 注册应用程序。

属性	描述
<code>OAuth2Options.Token.Enabled</code>	设为 True 以后用令牌端点。默认值：True。
<code>OAuth2Options.Token.URL</code>	令牌端点 URL 路径。默认值： <code>/sgc/oauth2/token</code> 。

应用注册

```
OAuth2Server->Apps->AddApp("MyApp", "",
    "my-client-id", "my-client-secret", 3600, true,
    TsgcOAuth2GrantTypes() << auth2ResourceOwnerPassword);
```

事件

事件	描述
<code>OnOAuth2Authentication</code>	当服务器接收到用户名和密码时触发。在此处验证凭据，并将 <code>Authenticated</code> 设置为 True 或 False。
<code>OnOAuth2AfterAccessToken</code>	服务器成功颁发访问令牌后触发，可用于日志记录。
<code>OnOAuth2AfterValidateAccessToken</code>	当客户端使用有效令牌发起请求时触发。

示例

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->Apps->AddApp("MyApp", "",
    "my-client-id", "my-client-secret", 3600, true,
    TsgcOAuth2GrantTypes() << auth2ResourceOwnerPassword);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;
void __fastcall TForm1::OAuth2ServerOAuth2Authentication(TObject *Sender,
    const UnicodeString User, const UnicodeString Password, bool &Authenticated)
{
    Authenticated = ValidateUser(User, Password);
}
void __fastcall TForm1::OAuth2ServerOAuth2AfterAccessToken(TObject *Sender,
    const UnicodeString Token, const UnicodeString RefreshToken, int ExpiresIn)
{
    Log("Token issued for user: " + Token);
}
```

何时使用

仅在以下情况下使用资源所有者密码凭据授权：

- 该客户端是高度可信的第一方应用程序。
- 其他授权类型（带 PKCE 的授权码）不可行。
- 迁移已直接收集用户凭据的旧版应用程序。

对于新应用程序，始终优先使用带 PKCE 的[授权码](#)授权方式。

OAuth2 服务器 | 刷新令牌授予

概述

刷新令牌授权允许客户端在不需要用户交互的情况下，以之前签发的刷新令牌换取新的访问令牌。[TsgcHTTP_OAuth2_Server](#) 验证刷新令牌并签发新的访问令牌（以及可选地签发新的刷新令牌）。这对于访问令牌有效期较短但客户端需要持续访问受保护资源的长期会话非常有用。

流程

1. 客户端使用 `grant_type=refresh_token&refresh_token=...&client_id=...&client_secret=...&scope=...` 向令牌端点发送 POST 请求
2. 服务器根据其存储的令牌验证刷新令牌。
3. 服务器触发 **OnOAuth2AfterRefreshToken**，并返回包含新 `access_token`、`token_type`、`expires_in` 以及可选新 `refresh_token` 的 JSON 响应。

配置

每个应用程序注册时启用刷新令牌。在 `Apps.AddApp` 中将 `RefreshToken` 参数设置为 `True`。

属性	描述
<code>OAuth2Options.Token.Enabled</code>	设为 <code>True</code> 以后用令牌端点。默认值： <code>True</code> 。
<code>OAuth2Options.Token.URL</code>	令牌端点 URL 路径。默认值： <code>/sgc/oauth2/token</code> 。
<code>RefreshToken (AddApp parameter)</code>	设置为 <code>True</code> 可为此应用程序签发刷新令牌。默认值： <code>True</code> 。

应用注册

注册应用时，将 `RefreshToken` 参数设置为 `True`，以便服务器在颁发访问令牌的同时颁发刷新令牌。

```
// RefreshToken = true (6th parameter)
OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
    "my-client-id", "my-client-secret", 3600, true,
    TsgcOAuth2GrantTypes() << auth2Code);
```

事件

事件	描述
<code>OnOAuth2AfterRefreshToken</code>	在服务器成功使用刷新令牌颁发新访问令牌后触发。可用于记录令牌轮转日志。
<code>OnOAuth2AfterAccessToken</code>	新访问令牌颁发后触发。

示例

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
    "my-client-id", "my-client-secret", 3600, true,
    TsgcOAuth2GrantTypes() << auth2Code);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;
void __fastcall TForm1::OAuth2ServerOAuth2AfterRefreshToken(TObject *Sender,
    const UnicodeString OldToken, const UnicodeString NewToken,
    const UnicodeString NewRefreshToken, int ExpiresIn)
{
    Log("Token refreshed. New token: " + NewToken);
}
```

令牌轮换

使用刷新令牌时，服务器可能会在颁发新访问令牌的同时颁发新的刷新令牌。旧刷新令牌随即失效。这被称为刷新令牌轮换，有助于防止令牌重放攻击。

重启后恢复令牌

若服务器在令牌仍有效时重启，您可以在启动服务器前使用 `Apps.AddToken` 恢复令牌。详情请参阅 [OAuth2 恢复访问令牌](#)。

OAuth2 服务器 | 设备授权授权 (RFC 8628)

概述

设备授权授权专为输入受限的设备（智能电视、IoT 设备、CLI 工具）设计，这些设备无法轻松显示浏览器或接受键盘输入。[TsgcHTTP_OAuth2_Server](#) 提供设备授权端点和验证页面，以使用户可以从辅助设备（如手机或计算机）授权该设备。

流程

1. 设备 POST 到 `/sgc/oauth2/device`，携带 `client_id=...&scope=...`
2. 服务器触发 **OnOAuth2DeviceAuthorization** 并返回包含 `device_code`、`user_code`、`verification_uri`、`expires_in` 和 `interval` 的 JSON 响应。
3. 设备向用户显示 `user_code` 和 `verification_uri`（例如："请前往 <https://example.com/sgc/oauth2/device/verify> 并输入代码：ABCD-1234"）。
4. 用户在辅助设备上导航至验证 URI 并输入 `user_code`。
5. 服务器触发 **OnOAuth2DeviceCodeVerification** 以验证 `user_code` 并授权设备。
6. 与此同时，设备以 `grant_type=urn:ietf:params:oauth:grant-type:device_code&device_code=...&client_id=...` 轮询令牌端点
7. 服务器返回 `authorization_pending`，直到用户授权。授权后，返回访问令牌。

配置

属性	描述
<code>OAuth2Options.DeviceAuthorization.Enabled</code>	设置为 True 以后用设备授权端点。默认值：False。
<code>OAuth2Options.DeviceAuthorization.URL</code>	设备授权端点的 URL 路径。默认值： <code>/sgc/oauth2/device</code>
<code>OAuth2Options.DeviceAuthorization.VerificationURL</code>	用户输入验证码的页面 URL 路径。默认值： <code>/sgc/oauth2/device/verify</code>
<code>OAuth2Options.DeviceAuthorization.ExpiresIn</code>	设备码的生命周期（秒）。默认值：600（10 分钟）。
<code>OAuth2Options.DeviceAuthorization.Interval</code>	设备应使用的最小轮询间隔（秒）。默认：5。

事件

事件	描述
<code>OnOAuth2DeviceAuthorization</code>	当设备请求设备码时触发。允许自定义响应或记录请求。
<code>OnOAuth2DeviceCodeVerification</code>	当用户在设备验证页面提交验证码时触发。验证 <code>user_code</code> 并授权或拒绝该设备。
<code>OnOAuth2AfterAccessToken</code>	服务器向设备颁发访问令牌后触发。

示例

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->OAuth2Options->DeviceAuthorization->Enabled = true;
OAuth2Server->OAuth2Options->DeviceAuthorization->ExpiresIn = 600;
OAuth2Server->OAuth2Options->DeviceAuthorization->Interval = 5;
OAuth2Server->Apps->AddApp("DeviceApp", "",
    "device-client-id", "device-client-secret", 3600, true,
    TsgcOAuth2GrantTypes() << auth2Code);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;
void __fastcall TForm1::OAuth2ServerOAuth2DeviceCodeVerification(TObject *Sender,
    const UnicodeString UserCode, bool &Authorized)
{
    Authorized = true;
}
void __fastcall TForm1::OAuth2ServerOAuth2AfterAccessToken(TObject *Sender,
    const UnicodeString Token, const UnicodeString RefreshToken, int ExpiresIn)
{
    Log("Device authorized. Token: " + Token);
}
```

轮询行为

在用户尚未授权设备期间，设备按配置的间隔轮询令牌端点。服务器响应：

- authorization_pending -- 用户尚未完成授权。
- slow_down -- 设备轮询过于频繁。设备应将间隔增加 5 秒。
- expired_token -- 设备代码已过期。设备必须重新启动流程。
- access_denied——用户拒绝了授权。

用户授权设备后，下一次轮询将返回访问令牌。

OAuth2 服务器 | 令牌撤销 (RFC 7009)

概述

令牌撤销允许客户端通知 `TsgcHTTP_OAuth2_Server` 某个先前颁发的访问令牌或刷新令牌已不再需要。服务器将使该令牌失效，使其不再能用于访问受保护资源。
当用户注销、应用程序卸载或令牌可能已泄露时，此功能非常有用。

端点

使用以下参数 POST 到 `/sgc/oauth2/revoke`：

参数	描述
<code>token</code>	要撤销的令牌（必填）。
<code>token_type_hint</code>	关于令牌类型的可选提示： <code>access_token</code> 或 <code>refresh_token</code> 。

行为

根据 [RFC 7009](#)，无论令牌是否被找到或成功吊销，服务器始终返回 HTTP 200 OK。这可以防止令牌存在性泄露——客户端无法通过观察响应状态来确定令牌是否有效。
当刷新令牌被吊销时，相关联的访问令牌也可能失效（具体取决于实现）。当访问令牌被吊销时，除非显式吊销，否则相关联的刷新令牌仍然有效。

配置

属性	描述
<code>OAuth2Options.Revocation.Enabled</code>	设置为 True 以后用撤销端点。默认值：False。
<code>OAuth2Options.Revocation.URL</code>	吊销端点的 URL 路径。默认值： <code>/sgc/oauth2/revoke</code>

事件

事件	描述
<code>OnOAuth2AfterRevokeToken</code>	令牌撤销尝试后触发。提供令牌值、 <code>token_type_hint</code> 以及指示令牌是否成功失效的 <code>Revoked</code> 参数。可用于记录撤销活动。

示例

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->OAuth2Options->Revocation->Enabled = true;

OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
    "my-client-id", "my-client-secret", 3600, true,
    TsgcOAuth2GrantTypes() << auth2Code);
```

```
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;

void __fastcall TForm1::OAuth2ServerOAuth2AfterRevokeToken(TObject *Sender,
    const UnicodeString Token, const UnicodeString TokenTypeHint, bool &Revoked)
{
    Log("Token revoked: " + Token + " (type: " + TokenTypeHint + ")");
    Revoked = true;
}
```

客户端请求示例

客户端通过向撤销端点发送 POST 请求来撤销令牌：

```
// Client-side: revoke an access token
HttpClient->Post("https://example.com/sgc/oauth2/revoke",
    "token=eyJhbGciOiI...&token_type_hint=access_token");
```


OAuth2 服务器 | 令牌内省 (RFC 7662)

概述

令牌内省允许资源服务器查询 [TsgcHTTP_OAuth2_Server](#)，以确定访问令牌或刷新令牌的活跃状态和元数据。当资源服务器需要在本本地解码令牌的情况下验证令牌时，这非常有用。

端点

POST 到 `/sgc/oauth2/introspect`，携带以下参数：

参数	描述
<code>token</code>	要自省的令牌（必填）。
<code>token_type_hint</code>	关于令牌类型的可选提示： <code>access_token</code> 或 <code>refresh_token</code> 。

响应

内省端点返回包含以下字段的 JSON 对象：

字段	描述
<code>active</code>	布尔值。如果令牌当前有效（有效且未过期或撤销），则为 <code>True</code> 。
<code>scope</code>	与令牌关联的作用域。
<code>client_id</code>	请求令牌的应用程序的客户端标识符。
<code>token_type</code>	令牌的类型（例如 <code>Bearer</code> ）。
<code>exp</code>	令牌的过期时间（Unix 时间戳）。
<code>username</code>	与令牌关联的用户名（如适用）。

如果令牌无效、已过期或已吊销，响应将仅包含 `{"active": false}`。

配置

属性	描述
<code>OAuth2Options.Introspection.Enabled</code>	设置为 <code>True</code> 以后用内省端点。默认值： <code>False</code> 。
<code>OAuth2Options.Introspection.URL</code>	自省端点 URL 路径。默认： <code>/sgc/oauth2/introspect</code>

事件

事件	描述
<code>OnOAuth2AfterIntrospectToken</code>	在令牌内省请求后触发。提供令牌值和可修改的 <code>IsActive</code> 参数，以覆盖默认验证结果。可用于自定义验证逻辑和审计。

示例

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->OAuth2Options->Introspection->Enabled = true;

OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
    "my-client-id", "my-client-secret", 3600, true,
    TsgcOAuth2GrantTypes() << auth2Code);

Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;

void __fastcall TForm1::OAuth2ServerOAuth2AfterIntrospectToken(TObject *Sender,
    const UnicodeString Token, bool &IsActive)
{
    Log("Token introspected: " + Token + " Active: " + BoolToStr(IsActive));
}
```

资源服务器请求示例

资源服务器通过向自省端点发送 POST 请求来验证令牌：

```
// Resource server: introspect a token
UnicodeString Response = HTTPClient->Post("https://example.com/sgc/oauth2/introspect",
    "token=eyJhbGciOiIuLj0token_type_hint=access_token");
// Response: {"active":true,"scope":"read write","client_id":"my-client-id",...}
```

OAuth2 服务器 | DPoP 验证 (RFC 9449)

概述

DPoP（持有证明演示）依据 [RFC 9449](#) 使 [TsgcHTTP_OAuth2_Server](#) 能够要求发送方约束令牌。启用 DPoP 后，服务器验证 DPoP 证明 JWT，将令牌绑定到客户端的 JWK 指纹，并返回 `token_type: DPoP` 而非 `Bearer`。这确保访问令牌只能由最初获取它的客户端使用，从而防止令牌被盗和重放攻击。

工作原理

1. 客户端在令牌请求中包含带有签名 JWT 证明的 DPoP 标头。
2. 服务器验证证明 JWT：
 - `typ` 必须为 `dpop+jwt`
 - 算法 (`alg`) 必须为非对称算法（如 ES256、RS256）
 - `htm` (HTTP 方法) 和 `htu` (HTTP URI) 必须与请求匹配
 - `iat` (签发时间) 必须是最近的（新鲜度检查）
 - 签名通过证明头中嵌入的 JWK 进行验证
3. 服务器提取 JWK 指纹并将其绑定到签发的令牌。
4. 令牌响应包含 `token_type: DPoP` 而非 `Bearer`。
5. 在后续资源请求时，服务器验证 DPoP 证明，包括 `ath` 声明（访问令牌哈希），以确保证明绑定到正确的令牌。

配置

属性	描述
<code>OAuth2Options.DPoP</code>	布尔值。设置为 <code>True</code> 以后用 DPoP 绑定令牌验证。默认值： <code>False</code> 。

签名验证

服务器从 DPoP 证明头中包含的 JWK 重建 EC 或 RSA 公钥，并验证 JWT 签名。仅接受非对称算法（ES256、ES384、ES512、RS256、RS384、RS512、PS256、PS384、PS512），对称算法（HS256 等）将被拒绝。

随机数支持

服务器可以在令牌响应中发出 `DPoP-Nonce` 标头。当存在时，客户端必须在后续 DPoP 证明 JWT 的 `nonce` 声明中包含该 `nonce` 值。这提供了额外的重放保护层。

事件

事件	描述
<code>OnOAuth2ValidateDPoP</code>	在标准 DPoP 验证完成后触发。允许自定义验证逻辑，例如检查额外的声明或强制执行策略。将 <code>Valid</code> 设为 <code>False</code> 可拒绝该证明。

示例

```
TsgcHTTP_OAuth2_Server *OAuth2Server = new TsgcHTTP_OAuth2_Server(this);
OAuth2Server->OAuth2Options->DPoP = true;
OAuth2Server->Apps->AddApp("MyApp", "http://127.0.0.1:8080",
    "my-client-id", "my-client-secret", 3600, true,
    TsgcOAuth2GrantTypes() << auth2Code);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2 = OAuth2Server;
void __fastcall TForm1::OAuth2ServerOAuth2ValidateDPoP(TObject *Sender,
    const UnicodeString DPoPProof, const UnicodeString AccessToken,
    const UnicodeString Method, const UnicodeString URL, bool &Valid)
{
    Log("DPoP proof validated for: " + Method + " " + URL);
    Valid = true;
}
```

无效证明拒绝

当 DPoP 证明验证失败时，服务器返回 HTTP 400 错误响应：

```
{"error": "invalid_dpop_proof", "error_description": "..."}

```

常见的拒绝原因包括：

- DPoP 标头缺失或格式错误。
- JWT 签名无效。
- 使用了对称算法（仅允许使用非对称算法）。
- htm 或 htu 声明不匹配。
- 已过期或未来的 iat 声明。
- 资源请求上 ath（访问令牌哈希）不匹配。
- 当服务器需要时，无效或缺少 nonce。

TsgcHTTP_OAuth2_Server_Provider

此组件允许您集成外部 OAuth2 提供商（如 Azure AD、Google、Facebook）。

简介

此组件允许您在服务器组件（如 HTTP 服务器）中集成外部 OAuth2 提供商（如 Azure AD、Google、Facebook 等），从而使用户可以通过 Azure AD 凭据登录，若验证成功，HTTP 服务器可提供对受保护资源的访问。

服务器组件有一个名为 `Authorization.OAuth.OAuth2Provider` 的属性，您可以在其中分配 `TsgcHTTP_OAuth2_Server_Provider` 的实例，因此如果启用了身份验证且 `OAuth2Provider` 属性附加到 `OAuth2` 提供程序服务器组件，则 `WebSocket` 和 `HTTP` 请求需要 `Cookie/承载令牌` 才能被处理，否则连接将自动关闭。

```
TsgcHTTP_OAuth2_Server_Provider *OAuth2Provider = new TsgcHTTP_OAuth2_Server_Provider(this);
Server->Authentication->Enabled = true;
Server->Authentication->OAuth->OAuth2Provider = OAuth2Provider;
```

注册 OAuth2 提供商

在启动服务器之前，必须配置服务器将用于身份验证的 OAuth2 提供程序。使用 `RegisterProvider` 方法配置 OAuth2 提供程序，此方法具有以下参数：

- **Name**：提供程序的名称，可以是任意名称，仅用于后续标识该提供程序。
- **ClientId**：公共客户端 ID，此值由 OAuth2 提供方提供。
- **ClientSecret**：私有客户端密钥（必须保密），此值由 OAuth2 提供商提供。
- **AuthorizeURL**：OAuth2 客户端将重定向到的登录 URL（连接使用 Web 浏览器）。
- **TokenURL**：服务器用于验证成功授权后所提供令牌的 URL（连接为服务器到服务器）。
- **Scope**：是范围/范围列表的值。
- **URL**：用于重定向到授权 URL 的 HTTP 服务器地址。
- **CallbackURL**：在 OAuth2 提供商中配置的 URL，用于处理成功授权后 OAuth2 服务器发送的响应。

示例：要配置 Azure AD，需要一个租户 ID，该 ID 被添加到 OAuth2 URL 中，还需要 `ClientId`、`ClientSecret`、`Scope` 和 `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'
);
```

要删除现有提供商，请使用方法 `UnRegisterProvider`。

参考

- 属性 — 组件上有 2 个可用属性。
- 方法 — 15 个公开方法。
- 事件 — 可处理 4 个事件。

指南

- **QuickStart**
 - [OAuth2 Provider Azure AD](#)
- **身份验证**
 - [OAuth2 提供方私有端点](#)
 - [OAuth2 提供方身份验证](#)
 - [OAuth2 提供方请求](#)

OAuth2 提供商 | Azure AD

Azure AD 使用以下 OAuth2 授权 URL

授权：https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/authorize

Token：https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token

<tenant-id> 必须替换为您自己的值。

创建 OAuth2 配置时，必须配置服务器回调 URL，Azure 将使用此 URL 在成功授权后向您的服务器发送响应。

示例：以下是如何注册 Azure AD 提供商的简单示例。

Azure AD 提供的值

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

如何注册 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 提供商 | 私有端点

每次服务器收到 HTTP 请求时，都会调用 **OnOAuth2IsPrivateEndpoint** 事件，询问该端点是否为私有端点。默认情况下不是私有端点。

```
void OnOAuth2IsPrivateEndpoint(TObject *Sender, const string aEndpoint, ref bool IsPrivate)
{
    if (aEndpoint == "/private")
    {
        IsPrivate = True;
    }
}
```


OAuth2 提供商 | 身份验证

OAuth2 Provider 服务器组件允许您使用外部 OAuth2 提供商（如 Azure AD 或 Google）进行认证，以访问服务器的受保护资源。**示例：**您可以配置 HTTP 服务器以允许用户使用 Azure 凭据登录；如果登录成功，这些用户将被允许访问服务器的受保护资源。

认证过程在服务器端完成，OAuth2 令牌不与客户端共享，这意味着当用户使用 Azure 等方式登录时，如果认证成功，Azure 会返回一个访问令牌，允许您向 Azure 服务器发送请求以获取用户配置文件、邮件等信息（取决于范围）。此访问令牌**不会与客户端（例如 Web 浏览器）共享**；服务器不会将访问令牌返回给客户端，而是创建一个随机 ID，该 ID 在内部与访问令牌关联，因此每当客户端（Web 浏览器）想要调用 OAuth2 服务器时，使用此公开 ID，服务器则使用该 ID 获取 OAuth2 访问令牌来代理 HTTP 请求。

以下是 OAuth2 身份验证工作原理的示例。该示例将使用以下链接中描述的 Azure AD 配置：[OAuth2 提供商 Azure AD](#)。

启动服务器

服务器在 localhost 的 443 端口上开始监听。sgcWebSockets HTTP 服务器已链接至 OAuth2 服务器提供程序组件，且 Authentication 属性已启用。
在服务器启动之前，使用以下方法调用注册 Azure OAuth2 提供程序。

```
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'  
);
```

用户登录

用户打开新的 Web 浏览器并访问 '/login' 端点。
服务器检测到使用 '/login' 端点通过 Azure 提供商登录，因此将重定向到

<https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/authorize>

OAuth2 身份验证流程开始。

OAuth2 身份验证

用户被重定向至 OAuth2 服务器身份验证端点，此时必须使用凭证登录并接受 OAuth2 应用程序的条款。

如果授权成功，Azure AD 将代码发送到 URL

<https://localhost/callback>

验证 OAuth2 代码

现在，服务器收到了来自 Azure 的代码，它将在内部（从服务器到服务器）连接 Azure 以验证该令牌是否正确（并防止有人尝试入侵服务器）。

服务器连接到

<https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/token>

传入接收到的 code 和 clientsecret 等参数，验证成功后，Azure 将返回访问令牌，该令牌可用于访问 Azure 受保护资源，例如读取个人资料、电子邮件等。

成功的访问令牌

当服务器成功收到 AccessToken 时，将调用 **OnOAuth2ProviderTokenValid** 事件，您可以在这里配置 AccessToken 的存储方式（如需存储），通过参数类 `TsgcHTTPOAuth2ProviderToken` 进行访问。

AccessToken：Azure 返回的 OAuth2 令牌

ID：作为 cookie 存储的公共标识符。

在此事件中，您可以配置身份验证成功后的操作，例如：如果您想将用户重定向到私有 URL，请使用以下方式。

```
Response.Redirect.URL := 'https://localhost/private';
```

向 Azure 发送请求

现在，您可以使用存储为 Cookie 的公共 ID 向 Azure 服务器发送请求。

示例：若要读取配置文件数据，请使用以下方法。

```
Get('ID', 'https://graph.microsoft.com/v1.0/me');
```

其中 ID 是公共 ID 标识符。

OAuth2 提供者 | 请求

身份验证成功后，您可以使用存储为 Cookie 的公共 ID 令牌向 OAuth2 受保护服务器发送请求。

OAuth2 Provider 服务器组件提供多种发送 HTTP 请求的方法：GET、POST、DELETE.....

您可以将令牌作为参数传入，或者在使用 Indy 服务器组件时传入 RequestInfo 类。

```
void OnCommandGet(TidContext *AContext, TidHttpRequestInfo *ARequestInfo, TidHttpResponseInfo *AResponseInfo)
{
    if (ARequestInfo->Document == "/private"
    {
        // return OAuth2 profile data
        AResponseInfo->ContentText = OAuth2Provider->Get(ARequestInfo, "https://graph.microsoft.com/v1.0/me");
        AResponseInfo->ContentType = "application/json";
        AResponseInfo->ResponseNo = 200;
    }
    else
    {
        AResponseInfo->ResponseNo = 404;
    }
}
```

头部

- **alg**：用于签署令牌的算法。
- **typ**：令牌类型，始终为 JWT。

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

载荷

- **iss**：签发者，即生成并签发 JWT 的实体。
- **sub**：主体，即此令牌所标识的实体。
- **aud**：受众，即此 JWT 的目标受众。
- **exp**：到期时间，UNIX 格式的时间戳，在此时间之后令牌不应被接受。
- **iat**：签发时间，指定令牌的签发日期。

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

签名

签名使用编码后的标头、编码后的载荷、密钥和加密算法创建。

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4uRG9lIiwiaWF0IjoxNTE2MjM5MjM5Lm50

DIyf0.Sf1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_ad0ssw5c

支持的算法

以下算法同时受客户端和服务端 JWT 组件支持。

- HS256
- HS384
- HS512
- RS256
- RS384
- RS512
- ES256
- ES384
- ES512

需要 **OpenSSL 库** 才能签名和验证 JWT。

组件

- **TsgcHTTP_JWT_Client** : JWT 客户端, 允许您对 JWT 进行编码和签名, 并在 **HTTP** 和 **WebSocket** 协议中作为授权头发送。
- **TsgcHTTP_JWT_Server** : JWT 服务器, 允许解码和验证作为 **HTTP** 和 **WebSocket** 协议中的 Authorization 标头接收的 JWT。

** JWT 组件至少需要 Indy 版本 10.6.0.5169 或 sgcWebSockets Enterprise 版。*

TsgcHTTP_JWT_Client

配置使用 JWT 客户端时将使用的 openssl 库。

简介

TsgcHTTP_JWT_Client 组件允许您对 JWT 令牌进行编码和签名，并将其附加到 [WebSocket 客户端](#) 或 [HTTP/2 客户端](#)，令牌将自动作为 Authorization Bearer Token 请求头发送。

配置

您可以在 **JWTOptions** 属性中配置 JWT 值，其中有两个主要属性：**Header** 和 **Payload**，只需为每个必填属性设置相应的值即可。

若签名使用私钥（RS 和 ES 算法）加密，请在算法的 **PrivateKey** 属性中设置相应值。
如果签名使用密钥（HS 算法）加密，请在算法的 **Secret** 属性中设置该值。

OpenSSL 选项

配置使用 JWT 客户端时将使用的 openssl 库。

OpenSSL_Options：OpenSSL 库的配置。

APIVersion：允许定义将使用哪个 OpenSSL API 版本。

oslAPI_1_0：使用 API 1.0 OpenSSL，这是 Indy 支持的最新版本

oslAPI_1_1：使用 API 1.1 OpenSSL，需要我们的自定义 Indy 库，允许使用支持 TLS 1.3 的 OpenSSL 1.1.1 库。

oslAPI_3_0：使用 API 3.0 OpenSSL，需要我们的自定义 Indy 库，允许使用支持 TLS 1.3 的 OpenSSL 3.0.0 库。

LibPath：在此处您可以配置 openssl 库的位置

oslpNone：这是默认值，OpenSSL 库应与二进制文件位于同一文件夹中，或位于已知路径中。

oslpDefaultFolder：自动设置所有 IDE 版本应查找 OpenSSL 库的路径。

oslpCustomFolder：如果选择此选项，请在属性 LibPathCustom 中定义完整路径。

LibPathCustom：当 LibPath = oslpCustomFolder 时，在此处定义 openssl 库所在的完整路径。

UnixSymLinks：启用或禁用在 Unix 系统下加载符号链接（默认启用，OSX64 除外）：

oslsSymLinksDefault：默认情况下启用，但在 OSX64 下除外（MacOS Monterey 之后，不带版本号尝试加载库会失败）。

oslsSymLinksLoadFirst：先加载符号链接，然后再尝试加载版本库。

oslsSymLinksLoad：在尝试加载版本库后加载符号链接。

oslsSymLinksDontLoad：不加载符号链接。

自定义标头

Header 和 Payload 属性包含生成 JWT 常用的标头，但您也可以通过调用 **AddKeyValue** 方法并传入键和值作为参数来添加更多标头。

示例：如果您想在 JWT 头部添加带有您姓名的新记录，请使用以下方法：

```
Header->AddKeyValue("name", "John Smith");
```

配置属性后，要生成 JWT，只需调用方法 **Sign**，它将返回 JWT 的值。

WebSocket 客户端与 JWT

[TsgcWebSocketClient](#) 支持在连接 WebSocket 服务器时使用 JWT，只需创建一个新的 JWT 客户端并将其赋值给 **Authentication.Token.JWT** 属性即可。

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient->URL = "ws://www.esegece.com:2052";

TsgcHTTP_JWT_Client oJWT = new TsgcHTTP_JWT_Client();
oJWT->JWTOptions->Header.alg = jwtRS256;
oJWT->JWTOptions->Payload.sub = "1234567890";
oJWT->JWTOptions->Payload.iat = 1516239022;

oClient->Authentication->Enabled = true;
oClient->Authentication->URL->Enabled = false;
oClient->Authentication->Token->Enabled = true;
oClient->Authentication->Token->JWT = oJWT;
oClient->Active = true;
```

HTTP 客户端和 JWT

[TsgcHTTP2Client](#) 和 [TsgcHTTP1Client](#) 允许在连接到 HTTP/2 服务器时使用 JWT，只需创建一个新的 JWT 客户端并分配给 **Authentication.Token.JWT** 属性即可。

```
TsgcHTTP2Client oHTTP = new TsgcHTTP2Client();

TsgcHTTP_JWT_Client oJWT = new TsgcHTTP_JWT_Client();
oJWT->JWTOptions->Header->alg = jwtRS256;
oJWT->JWTOptions->Payload->sub = "1234567890";
oJWT->JWTOptions->Payload->iat = 1516239022;

oHTTP->Authentication->Token->JWT = oJWT;
oHTTP->Get("https://your.api.com");
```

过期时间

授权令牌可以在每次发送 HTTP 请求时**重新创建**，也可以**多次复用直至过期**。

示例：使用令牌调用 Apple APNs，要求令牌至少重用 20 分钟，最多 1 小时。使用属性 **RefreshTokenAfter** 设置令牌过期的秒数，例如 30 分钟后。

```
RefreshTokenAfter = 60 * 40.
```

创建 JWT 签名

您可以**手动创建 JWT 签名**，用于不使用 WebSocket 或 HTTP 协议的应用程序，或者如果您使用第三方应用程序的组件而只需要 JWT 令牌。

要获取 JWT 签名，只需创建 JWT 客户端的新实例并手动填写属性。当所有属性设置完毕后，调用方法 **Sign**，它将返回 JWT 令牌。

```
TsgcHTTP_JWT_Client oJWT = new TsgcHTTP_JWT_Client(this);  
// ... header  
oJWT->JWTOptions->Header->alg = jwtHS256;  
oJWT->JWTOptions->Algorithms->HS->Secret = "79F66F1E-E998-436B-8A0A-3E5DEFA4FD9E";  
// ... payload  
oJWT->JWTOptions->Payload->jti = "9B66FB94-B761-42B1-A1AF-3C44233DBE87";  
oJWT->JWTOptions->Payload->iat = 1630925658;  
oJWT->JWTOptions->Payload->iss = "2886EC7547B7BA6A9009";  
oJWT->JWTOptions->Payload->exp = 1630933158;  
// ... custom payload values  
oJWT->JWTOptions->Payload->ClearKeyValues;  
oJWT->JWTOptions->Payload->AddKeyValue("origin", "www->yourwebsite->com");  
oJWT->JWTOptions->Payload->AddKeyValue("ip", "69.39.46.178");  
// ... get JWT Token  
ShowMessage(oJWT->Sign());
```

参考

- 属性 — 组件上提供 2 个属性。
- 方法 — 2 个公共方法。
- 事件 — 可处理 0 个事件。

TsgcHTTP_JWT_Server

TsgcHTTP_JWT_Server 组件。

简介

TsgcHTTP_JWT_Server 组件允许您对 JWT 令牌进行**解码和验证**，令牌可以在使用 WebSocket 协议时通过 **WebSocket 握手**接收，或在使用 HTTP 协议时作为 **HTTP 标头**接收。

配置

您可以在组件的 **JWTOptions** 属性中配置以下属性：

如果使用公钥（RS 和 ES 算法）验证签名，请在算法的 **PublicKey** 属性中设置该值。
如果使用密钥（HS 算法）验证签名，请在算法的 **Secret** 属性中设置值。

若要验证 JWT 令牌，只需将 **TsgcHTTP_JWT_Server** 实例附加到 WebSocket/HTTP 服务器的 **Authentication.JWT.JWT** 属性即可。

```
TsgcWebSocketHTTPServer oServer = new TsgcWebSocketHTTPServer();
oServer->Port = 80;
TsgcHTTP_JWT_Server oJWT = new TsgcHTTP_JWT_Server();
oJWT->JWTOptions->Algorithms->RS->PublicKey->Text = "public key here";
oServer->Authorization->Enabled = true;
oServer->Authorization->JWT->JWT = oJWT;
oServer->Active = true;
```

Checks 属性允许您启用对 JWT 载荷的某些检查，默认情况下检查签发日期是否有效。

参考

- 属性 — 组件上有 2 个可用属性。
- 方法 — 3 个公开方法。
- 事件 — 7 个可处理的事件。

WebAuthn

WebAuthn（Web 身份验证）是由万维网联盟（W3C）和 FIDO 联盟开发的 Web 标准，旨在实现 **安全的、无密码的网络身份验证**。它是更广泛的 FIDO2 框架的一部分，旨在 **减少对传统密码的依赖**，因为传统密码往往容易受到网络钓鱼、凭据填充及其他攻击的威胁。

WebAuthn 的核心在于**允许用户使用公钥密码学进行身份验证**。用户无需使用用户名和密码，而是向 **Web 应用程序**（信赖方）注册一对唯一的公私钥。私钥安全地存储在身份验证器上（如硬件安全密钥、智能手机或内置生物识别设备），而公钥则存储在服务器上。

在身份验证期间，服务器发出一个必须由用户私钥签名的挑战。签名后的挑战被返回并使用存储的公钥进行验证，确保响应的完整性和来源。这种方法防止凭证被截获或重用。

WebAuthn 支持多种身份验证器和设备，使其对开发者和用户都非常灵活。当与 PIN 码或生物特征等其他因素结合使用时，它还支持多因素身份验证（MFA），在不牺牲可用性的情况下显著提高安全性。

服务器

- **TsgcWSAPIServer_WebAuthn**：该组件提供了一个简单但功能强大的解决方案，用于在您的 Web 应用程序中实现 WebAuthn 信赖方服务器，实现无密码身份验证。服务器已通过使用 Conformance Self-Validation Testing 工具的完整 **FIDO 合规性测试**。

Client

- **Javascript 客户端**：在注册和身份验证期间充当用户、浏览器和认证器之间的桥梁。

TsgcWSAPIServer_WebAuthn

仅 OpenSSL 3。

简介

TsgcWSAPIServer_WebAuthn 组件提供了一个简单而强大的解决方案，用于实现 WebAuthn 依赖方服务器，在 Web 应用程序中实现无密码身份验证。一个 WebAuthn 应用程序由一个处理服务器端注册和身份验证的 WebAuthn 服务器，以及通常是 JavaScript 应用程序的客户端应用程序组成。

WebAuthn 要求使用安全连接（SSL/TLS），因此必须部署 [OpenSSL](#) 库并在服务器上配置。

仅支持 OpenSSL 3.0.0+ API，因此之前的 OpenSSL 版本可能无法正常工作。

配置

TsgcWSAPIServer_WebAuthn 必须使用 `Server` 属性附加到 HTTP 服务器，即 [TsgcWebSocketHTTP_Server](#) 或 [TsgcWebSocketServer_HTTPAPI](#)。您可以配置处理注册和身份验证选项的服务器端点，以及 WebAuthn 选项（如支持的算法、来源等）。

端点选项

在此处您可以配置处理 HTTP/JavaScript 请求的服务器端点，以使用 WebAuthn 作为验证器。该组件已配置默认端点，但您可以更改所有端点以满足您的需求。

- **AuthenticationOptions**：默认为 `/sgcWebAuthn/Authentication/Options`
- **AuthenticationVerify**：默认为 `/sgcWebAuthn/Authentication/Verify`
- **RegistrationOptions**：默认为 `/sgcWebAuthn/Registration/Options`
- **RegistrationVerify**：默认为 `/sgcWebAuthn/Registration/Verify`
- **Webauthn**：包含默认使用的 JavaScript 库。您可以禁用此属性并使用您自己的 WebAuthn 库。
- **测试**：默认情况下禁用，仅用于测试 WebAuthn 功能。

示例：如果您的服务器正在监听域 `www.test.com`，则默认情况下，身份验证选项请求将为 `http://www.test.com/sgcWebAuthn/Authentication/Options`

WebAuthn 选项

在此属性中，您可以配置 WebAuthn 服务器组件的主要选项。

- **RelyingParty**：一个**必填**属性，必须定义**服务器的 DNS 名称**。示例：如果服务器运行在域名 `www.test.com` 上，则将此属性设置为 `"www.test.com"`。

WebAuthn 使用来源来强制执行同源策略约束，这对于防止网络钓鱼和跨站攻击至关重要。在 WebAuthn 注册和身份验证过程中，浏览器和身份验证器会严格验证来源。

- **Origins**：若请求可能来自不同来源，请使用 `Origin` 属性设置额外的来源。示例：若请求可能来自 `login.test.com` 和 `www.test.co.uk`，请在 `Origins` 属性中配置以下值：`https://login.test.com` 和 `https://www.test.co.uk`
- **TopOrigins**：通常，WebAuthn 依赖调用帧的来源（即调用 `navigator.credentials.create()` 或 `navigator.credentials.get()` 的帧）。然而，网页可以嵌入在 `iframe` 中，该 `iframe` 可能来自与顶级页面不同的

来源。这为滥用或点击劫持式攻击提供了可能性。为缓解此问题，WebAuthn Level 2 规范引入了 TopOrigin，您可以在其中定义 TopOrigins。

在 WebAuthn 中，crossOrigin 是一个布尔参数，指示 WebAuthn 操作是否在跨源上下文中执行，例如嵌入在与顶级浏览上下文不同源的 iframe 中。

引入此参数是为了帮助浏览器和身份验证器在嵌入式环境中安全处理身份验证请求。

- **AllowCrossOrigins**：如果为 true，表示允许来自跨源 iframe 的请求（例如，嵌入在 https://app.example.org 页面中的 https://auth.example.com iframe）。默认情况下禁用。

WebAuthn 支持多种 **加密算法**，用于公钥凭据的生成和验证。这些算法在凭据注册（通过 navigator.credentials.create()）和身份验证（通过 navigator.credentials.get()）期间使用，并通过非对称密钥对确保挑战的安全签名和验证。服务器默认配置了最常用的 **ES256 和 RS256** 算法。您可以随时通过 **Algorithms** 属性更改所支持的算法。支持以下算法：

- ES256
- ES384
- ES512
- RS256
- RS384
- RS512
- PS256
- PS384
- PS512
- RS1
- EdDSA

在 WebAuthn 中，**证明是一种可选机制**，允许验证器（例如设备或安全密钥）在凭据创建期间提供有关其制造商、型号和安全特性的信息。这些信息有助于依赖方（RP）决定是否信任该验证器。

不同的认证格式定义于此数据的结构和验证方式。三种常用格式为 android-key、packed 以及其他格式（如 fido-u2f、apple 或 none）。默认情况下，所有认证格式均已启用。您可以在下方找到受支持的认证格式列表：

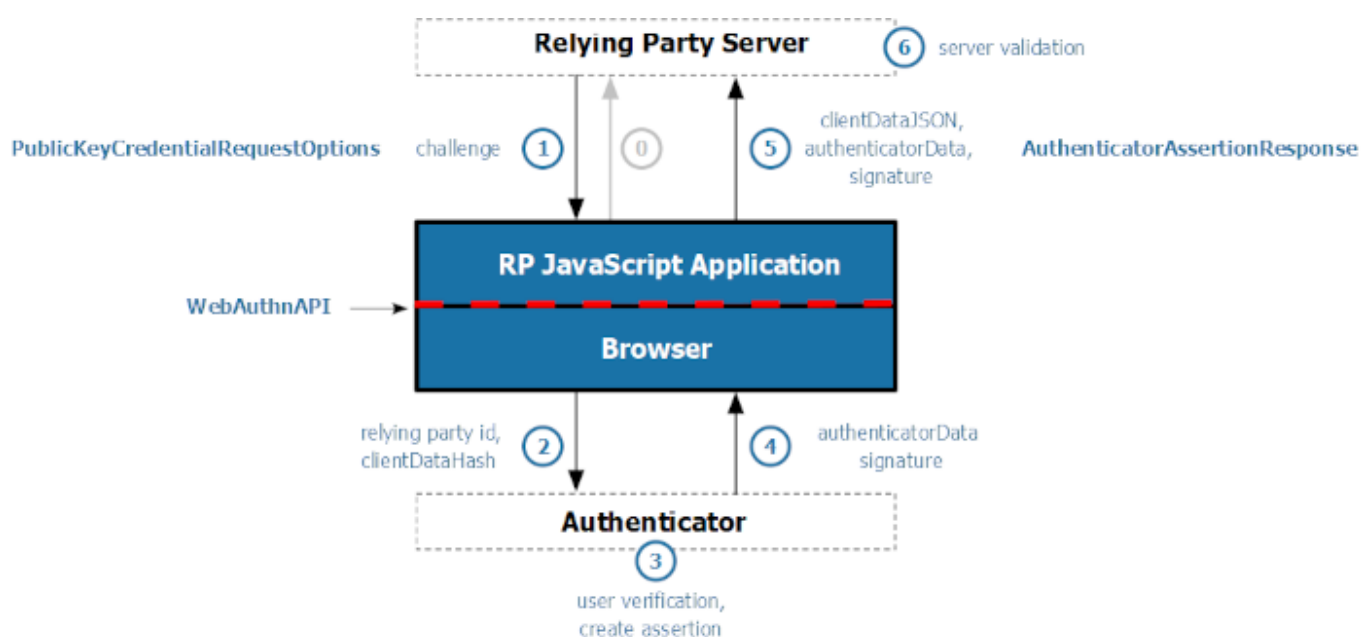
- **NoneAttestation**：在此情况下不返回任何证明数据。通过避免暴露设备标识符来优先保护用户隐私。常用于不关心设备来源的应用程序。
- **PackedAttestation**：是许多身份验证器使用的灵活紧凑格式。身份验证器返回证明证书和签名。可以是：**完整证明**：使用供应商提供的密钥和证书签名，或**自我证明**：使用凭证私钥签名。在不同平台上使用最广泛（例如 YubiKey、Windows Hello）。
- **TPMAttestation**：由配备可信平台模块（TPM）的设备使用。证明使用 TPM 中的密钥进行签名，并包含证书链。用于配备 TPM 芯片的企业台式机/笔记本电脑（例如 Windows 机器）。
- **AndroidKeyAttestation**：由具有 Android Keystore 的 Android 设备使用。密钥在硬件中生成，证明包含由设备制造商签发的证书链签名的信息。适用于具有硬件支持密钥库（TEE 或 StrongBox）的 Android 手机。
- **AppleAttestation**：由 Apple 平台身份验证器（如 Touch ID 和 Face ID）使用。证明由 Apple 内部 API 生成，包含特殊证书格式。在使用 Apple 生物特征的 Safari 上使用。
- **FidoU2FAttestation**：FIDO U2F 认证器使用的旧版证明格式。返回兼容 U2F 的证书和签名。用于支持 FIDO U2F 的较旧安全密钥（例如早期 YubiKeys）。

在 WebAuthn API 中，**AllowCredentials** 是身份验证过程（通过 navigator.credentials.get()）中使用的**可选字段**。它指定允许对特定依赖方（RP）进行用户身份验证的凭据 ID 列表。此机制允许 RP 控制哪些凭据被视为有效的登录尝试。credentials 属性包含以下字段：

- **AllowCredentials**：若启用（默认为 false），则指定允许对用户进行身份验证的凭据 ID 列表
- **ExcludeCredentials**：给定用户名，显示服务器组件中已存储的所有现有凭据。
- **Limit**：ExcludeCredentials 为 true 时将发送的最大凭据数量。

WebAuthn 协议

- **WebAuthn 注册**：服务器生成一个挑战并将其发送给客户端，客户端使用验证器（例如安全密钥或生物特征设备）创建密钥对。公钥被发回并由服务器存储，用于将来的认证。以下是有关注册流程事件的更多信息：
 - **WebAuthn 注册请求**
 - **WebAuthn 注册响应**
 - **WebAuthn 注册结果**
- **WebAuthn 身份验证**：服务器向客户端发送一个挑战值，客户端使用存储在验证器中、之前注册的私钥对其进行签名。服务器使用存储的公钥验证签名响应，从而确认用户身份。以下是有关身份验证流程事件的更多信息：
 - **WebAuthn 身份验证请求**
 - **WebAuthn 身份验证响应**
 - **WebAuthn 身份验证结果**
- **MDS**：FIDO 联盟元数据服务（MDS）是元数据声明的集中式存储库，供依赖方用于验证身份验证器证明并证明设备型号的真实性。



- **授权**：客户端可以在身份验证流程中向服务器请求承载令牌。此令牌可在以后用于打开新的 WebSocket 或 HTTP 连接，而无需使用通行密钥再次登录。
 - **WebAuthn 授权 WebSocket**
 - **WebAuthn 授权 HTTP**

参考

- 属性 — 组件上有 4 个可用属性。
- 方法 — 8 个公共方法。
- 事件 — 15 个可处理的事件。

WebAuthn | 注册

WebAuthn 注册涉及**客户端**（浏览器）、**身份验证器**（安全设备）和**依赖方**（RP；TsgcWSAPIServer_WebAuthn Delphi/CBuilder 服务器）。

注册选项

创建新用户凭据通常需要客户端使用新的 HTTP 请求向 TsgcWSAPIServer_WebAuthn 服务器中配置的**注册选项端点**发起注册流程。默认情况下，端点为 /sgcWebAuthn/Registration/Options，因此如果您的服务器正在监听域名 https://www.test.com，客户端应向 https://www.test.com/sgcWebAuthn/Registration/Options 发出新请求。

客户端发送新请求，将以下 JSON 作为有效负载传递（以 test 作为用户名）

```
{"username": "test", "algorithms": []}
```

服务器读取请求并返回带有新挑战的响应。

```
{
  "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
  }
}
```

响应返回以下数据：

- **challenge**：随机化的二进制数据，经 Base64 编码。
- **rp**：依赖方信息（您的 Web 应用/网站）。
- **user**：用户标识符和显示名称。
- **pubKeyCredParams**：允许的算法（例如 ES256、RS256）。
- **authenticatorSelection**：认证器类型的偏好设置（平台认证器或跨平台认证器、用户验证选项）。

- **timeout**：建议的超时值。
- **attestation**：证明传达偏好（"none"、"direct"、"indirect"）。

注册验证

客户端现已获取服务器响应，读取响应后，认证器将密码学数据返回给客户端 Web 浏览器。然后客户端向 TsgcWSAPIServer_WebAuthn 服务器上配置的注册验证端点（默认为 /sgcWebAuthn/Registration/Verify）发送新的 HTTP 请求，携带以下数据。

，因此如果您的服务器正在监听域名 <https://www.test.com>，客户端应向 <https://www.test.com/sgcWebAuthn/Registration/Verify> 发出新请求。

身份验证器将以下内容响应给浏览器中的 JavaScript：

- **id**：凭据 ID（身份验证器生成的密钥对的唯一 ID）。
- **rawId**：凭据 ID 的二进制形式。
- **type**：凭据类型（"public-key"）。
- **response**：包含 attestationObject 和 clientDataJSON。

以下是一个示例：

```
{
  "id": "yeA4BVRlrAfLG-KzqsL_rll4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
  "rawId": "yeA4BVRlrAfLG-KzqsL_rll4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
  "response": {
    "attestationObject": "o2NmbXRmcGFja2VkZ2F0dFN0....",
    "clientDataJSON": "eyJ0eXBlljoid2ViYXV0aG4uY3Jl....",
    "transports": [
      "nfc",
      "usb"
    ],
    "publicKeyAlgorithm": -7,
    "publicKey": "MFkwEwYHKoZIzj0CAQYIKoZIzj....",
    "authenticatorData": "SZYN5YgOjGh0NBcPZHgW4_k...."
  },
  "type": "public-key",
  "clientExtensionResults": {
    "credProps": {
      "rk": true
    }
  },
  "authenticatorAttachment": "cross-platform"
}
```

服务器读取来自客户端的 JSON 请求，并对公钥和凭据 ID 进行解码、验证和存储。

解码 attestationObject 和 clientDataJSON：

- 从 attestationObject 中提取公钥。
- 验证 clientDataJSON 中的挑战与服务器发送的原始挑战匹配。
- 确保正确的来源验证（myapp.example.com）。

存储凭证：

- 如果验证成功，将调用 **OnWebAuthnRegistrationSuccessful** 事件，您可以将凭据 ID 和提取的公钥安全存储在数据库中，以供日后身份验证使用。
- 如果发生任何错误，将调用事件 **OnWebAuthnRegistrationError**，您可以从中获取错误详情。

注册流程

有关注册流程的更多信息，请访问以下链接：

- [WebAuthn 注册请求](#)
- [WebAuthn 注册响应](#)
- [WebAuthn 注册结果](#)

WebAuthn 注册 | 请求

此注册选项请求对于安全引导 WebAuthn 注册至关重要。它要求服务器：

- 生成挑战，
- 查找或预配用户 ID，
- 返回有效的 `PublicKeyCredentialCreationOptions` 对象（包含 RP、用户、算法等）。

浏览器必须从服务器获取凭据创建选项以启动安全的 WebAuthn 凭据注册。这通常通过从浏览器向服务器端点（例如 `/sgcWebAuthn/Registration/Options`）发出 HTTPS POST 请求来完成，该请求会准备挑战、RP ID 和用户信息等参数。

请求体包含标识用户的数据，可能还包含配置偏好设置。通常格式如下：

```
{
  "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
}
```

WebAuthn 服务器组件在 `EndpointOptions.RegistrationOptions` 属性配置的端点上监听，处理浏览器获取 `PublicKeyCredentialCreationOptions` 的初始请求。当服务器收到新的 HTTP 请求时，会调用事件 **OnWebAuthnRegistrationOptionsRequest**，您可以访问客户端发送的请求，并通过将 `Accept` 参数设置为 `False` 来取消请求。

```
void __fastcall TForm1::OnWebAuthnRegistrationOptionsRequest(TObject *Sender, TsgcWebAuthn_RegistrationOptions_Re
{
    if (aRequest->Username == "anonymous")
        Accept = false;
}
```

WebAuthn 注册 | 响应

服务器响应客户端的注册选项请求（例如 POST /sgcWebAuthn/Registration/Options），返回如下所示的 JSON 载荷（二进制字段经过 base64url 编码）：

```
{
  "publicKey": {
    "rp": {
      "name": "esegece software",
      "id": "esegece.com"
    },
    "user": {
      "id": "c3ViamVjdC1pZA", // base64url-encoded ArrayBuffer (user handle)
      "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"
    }
  }
}
```

以下是字段说明：

- **rp**：依赖方信息：
 - **name**：友好显示名称（例如"Example Inc."）。
 - **id**：依赖方 ID（必须与站点来源匹配或为可注册后缀）。
- **user**：标识用户账户的信息：
 - **id**：唯一、稳定的字节数组（例如用户 UUID）。
 - **name**：唯一用户标识符（用户名或邮箱）。
 - **displayName**：面向用户的显示名称。
- **challenge**：密码学随机 nonce（16–64 字节），以 base64url 编码。
- **pubKeyCredParams**：依赖方愿意接受的算法：
 - 常见：-7 (ES256)、-257 (RS256)
- **timeout**：创建操作的可选超时时间（毫秒）。
- **attestation**："none"、"indirect"、"direct" 之一。
- **authenticatorSelection**：
 - **authenticatorAttachment**："platform"（内置）或 "cross-platform"（例如 YubiKey）。
 - **userVerification**："required"、"preferred" 或 "discouraged"。
 - **residentKey**："required"、"preferred" 或 "discouraged"。

在响应发送给客户端之前，将调用事件 **OnWebAuthnRegistrationOptionsResponse**，允许您自定义响应。

```
void __fastcall OnWebAuthnRegistrationOptionsResponse(TObject *Sender,
  TsgcWebAuthn_RegistrationOptions_Request* aRequest,
  TsgcWebAuthn_RegistrationOptions_Response* aResponse)
{
  if (aRequest->Username == "esegece.com") {
    aResponse->ExcludeCredentials->AddCredentialRecordFromJSON("json1.txt");
    aResponse->ExcludeCredentials->AddCredentialRecordFromJSON("json2.txt");
  }
}
```

```
} }
```

WebAuthn 注册 | 结果

目标是验证客户端返回数据的真实性和完整性，确保凭据绑定到预期用户，并安全地注册公钥凭据以供将来身份验证使用。

服务器必须按以下步骤验证客户端响应：

- 解析客户端响应：客户端向服务器发送如下响应。

```
{
  "id": "base64url-encoded credential ID",
  "rawId": "base64url-encoded ID bytes",
  "response": {
    "clientDataJSON": "base64url",
    "attestationObject": "base64url"
  },
  "type": "public-key"
}
```

- 服务器验证 clientDataJSON 和 attestationObject
- 验证证明声明（若已定义）
- 如果一切有效，则存储凭据，并可信任公钥用于将来的登录。

注册成功

如果客户端发送的响应有效，则调用 **OnWebAuthnRegistrationSuccessful** 事件，凭据记录可以安全地存储到数据库中以供未来登录验证使用。

```
void __fastcall OnWebAuthnRegistrationSuccessful(TObject *Sender,
  TsgcWebAuthn_Registration* aRegistration,
  TsgcWebAuthn_CredentialRecord* aCredentialRecord,
  bool &Accept)
{
  // Store in a DB
  DB->Credentials->Append();
  DB->Credentials->FieldByName("Credentials")->AsString = aCredentialRecord->AsJSON;
  DB->Credentials->Post();
}
```

注册错误

如果在验证客户端响应时出现任何错误，则调用事件 **OnWebAuthnRegistrationError**，您可以在参数 aError 中获取错误原因。

```
void __fastcall OnWebAuthnRegistrationError(TObject *Sender,
  TsgcWebAuthn_RegistrationVerify_Request* aRequest,
  TsgcWebAuthn_Registration* aRegistration,
  const String aError)
{
  Log("#webauthn_registration_error: " + aError);
}
```

WebAuthn | 身份验证

WebAuthn 身份验证允许用户使用之前注册的公钥凭据登录。它涉及使用用户注册时存储的公钥对已签名的质询进行验证。

身份验证选项

身份验证需要客户端通过向 TsgcWSAPIServer_WebAuthn 服务器中配置的**身份验证选项端点**发送新的 HTTP 请求来启动身份验证流程。默认情况下，该端点为 /sgcWebAuthn/Authentication/Options，因此，如果您的服务器监听的域名为 https://www.test.com，则客户端应向 https://www.test.com/sgcWebAuthn/Authentication/Options 发送新请求。

客户端通过 POST 将断言（认证响应）发送到服务器

```
{"username": "test", "user_verification": "preferred"}
```

服务器生成 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": {}
}
```

身份验证验证

身份验证要求客户端通过向 TsgcWSAPIServer_WebAuthn 服务器中配置的**身份验证验证端点**发起新的 HTTP 请求来启动身份验证流程。默认端点为 /sgcWebAuthn/Authentication/Verify，因此如果您的服务器监听域名 https://www.test.com，则客户端应向 URL https://www.test.com/sgcWebAuthn/Authentication/Verify 发起新请求。

浏览器提示用户使用其身份验证器（例如指纹识别、YubiKey）。身份验证器使用与凭据 ID 关联的私钥对挑战进行签名。返回的凭据包括：

- **rawId**：凭据 ID（base64url）。
- **response.authenticatorData**：来自认证器的元数据（二进制）。
- **response.clientDataJSON**：包含原始质询、来源和类型。
- **response.signature**：对 authenticatorData + hash(clientDataJSON) 的签名。

请参阅以下客户端请求的 JSON 示例：

```
{
  "id": "yeA4BVRlrAfLG-KzqsL_rlI4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
  "rawId": "yeA4BVRlrAfLG-KzqsL_rlI4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",

```

```
"response": {  
  "authenticatorData": "SZYN5YgOjGh0NBcPZHZgW4_krrmihjLHmVzzuoMdl2MFAAAABw",  
  "clientDataJSON": "eyJ0eXBldjoid2ViYXV0aG4uZ.....",  
  "signature": "MEQCIARqvvys8....",  
  "userHandle": "36b9d6a84204487382fee62e7e67a80d"  
},  
"type": "public-key",  
"clientExtensionResults": {},  
"authenticatorAttachment": "cross-platform"  
}
```

服务器读取来自客户端的请求，验证凭据已存储，并验证签名。如果签名有效，则调用事件 **OnWebAuthnAuthenticationSuccessful**。

身份验证流程

通过以下链接获取有关身份验证流程的更多信息：

- [WebAuthn 认证请求](#)
- [WebAuthn 身份验证响应](#)
- [WebAuthn 认证结果](#)

WebAuthn 认证 | 请求

当用户尝试登录时，浏览器向服务器发送请求，要求获取身份验证选项（也称为“断言选项”）。

```
{  
  "username": "alice@example.com"  
}
```

此请求允许服务器：

- 生成一个挑战值（安全的随机值）。
- 查找哪些凭据（公钥 ID）对该用户有效。
- 使用浏览器将在 `navigator.credentials.get()` 中使用的加密参数进行响应。

当 WebAuthn 服务器组件收到此请求时（通常通过浏览器向服务器端点（例如 `/sgcWebAuthn/Authentication/Options`）的 HTTPS POST 请求完成），事件 **OnWebAuthnAuthenticationOptionsRequest** 被调用，您可以在这里添加与用户名关联的凭据（如有）。

```
void __fastcall TForm1::OnWebAuthnAuthenticationOptionsRequest(  
    TObject *Sender,  
    TsgcWebAuthn_AuthenticationOptions_Request *aRequest,  
    TsgcWebAuthn_CredentialRecords &CredentialRecords,  
    bool &Accept)  
{  
    if (UserExistsInDB(aRequest->Username)) {  
        while (!EOF()) {  
            CredentialRecords.AddCredentialRecordFromJSON(RecordFromDB());  
            Next();  
        }  
    }  
}
```

WebAuthn 身份验证 | 响应

一旦浏览器（客户端）向服务器发送带有用户名（或类似标识符）的请求，服务器会回复客户端将用于通过浏览器的 `navigator.credentials.get()` API 开始认证流程的认证选项。

此响应提供浏览器使用用户的身份验证器（例如安全密钥、生物识别设备）生成 WebAuthn 身份验证断言所需的参数和约束。

示例 JSON 响应：

```
{
  "challenge": "Z3lVbWV5YXBpbmdvZG90IQ",
  "timeout": 60000,
  "rpId": "example.com",
  "allowCredentials": [
    {
      "type": "public-key",
      "id": "dXNlckNyZWJRJZA",
      "transports": ["usb", "nfc", "ble"]
    }
  ],
  "userVerification": "preferred"
}
```

以下是字段说明：

- **challenge**：加密挑战（随机生成），用于防止重放攻击。每次请求必须唯一。
- **timeout**：可选。浏览器在取消操作前应等待的时间（毫秒）。
- **rpId**：依赖方标识符，通常是您的域名。
- **allowCredentials**：用户之前注册的可接受凭据列表（凭据 ID）。
- **userVerification**：告知浏览器如何验证用户："required"、"preferred" 或 "discouraged"。
- **extensions（可选）**：从验证器请求的附加功能或数据。

在响应发送给客户端之前，调用 **OnWebAuthnAuthenticationOptionsResponse** 事件，允许您自定义响应。

WebAuthn 身份验证 | 结果

目标是验证浏览器提供的已签名断言，该断言证明用户拥有最初注册的私钥。这是安全登录用户的关键。用户与其认证器（例如指纹、安全密钥）交互后，浏览器会将包含认证结果的 POST 请求发送回服务器。以下是一个 JSON 示例：

```
{
  "id": "credential-id",
  "rawId": "base64url-encoded-credential-id",
  "type": "public-key",
  "response": {
    "clientDataJSON": "base64url",
    "authenticatorData": "base64url",
    "signature": "base64url",
    "userHandle": "optional"
  }
}
```

当服务器在配置的端点（例如 `/sgcWebAuthn/Authentication/Verify`）收到此请求时，必须验证以下步骤：

- 从数据库检索凭据：使用凭据 ID 查找用户已注册的公钥和元数据（例如签名计数器）。
- 解码并解析 `clientDataJSON`
- 验证身份验证器数据
- 验证签名

如果所有验证均正确，则身份验证成功，并调用事件 **OnWebAuthnAuthenticationSuccessful**。

若任何检查失败，将调用 **OnWebAuthnAuthenticationError** 事件并说明错误原因。

WebAuthn | MDS

元数据服务（MDS）是 **FIDO 联盟**提供的**集中式服务**，汇聚并发布通过 FIDO 认证计划认证的身份验证器的**元数据声明**。这些声明包含有关身份验证器的详细安全、合规性和操作信息。

- 该服务端点通常称为 **MDS3**（元数据服务协议第三版）。
- 依赖方（例如实现 WebAuthn 的网站或应用程序）从 MDS 中**检索元数据声明**，以便就身份验证器做出明智的信任决策。

MDS 为使用 WebAuthn 的依赖方添加了重要的信任和安全验证层：

- **验证器验证**：启用对验证器是否符合 FIDO 标准的验证，并帮助验证 WebAuthn 证明中提供的 AAGUID。
- **泄露与吊销检测**：提供有关已泄露或已吊销验证器的最新信息，允许依赖方阻止不安全的设备。
- **安全保障**：帮助执行安全策略，例如仅允许满足特定 FIDO 认证级别或用户验证强度的验证器。
- **互操作性**：确保跨不同浏览器、平台和使用不同验证器的设备之间行为一致且安全预期一致。

配置

您可以使用 `WebAuthnOptions.MDS` 属性配置 MDS 的使用，以下是主要属性：

- **Enabled**：若为 `true`（默认值），WebAuthn 请求将根据已配置的 MDS 文件进行验证。
- **MDS_FileName**：MDS 文件的存储路径。可从以下 URL 下载：<https://mds3.fidoalliance.org/>
- **RootCert_FileName**：存储根证书的路径。必须定义此路径才能验证证书链。可从 <https://valid.r3.roots.globalsign.com/> 下载。
- **Leaf_CertificateCRL**：如果为 `true`（默认为 `false`），将根据 CRL（证书吊销列表）验证 Blob 的叶证书。
- **CRL_FileName**：CRL 文件的存储路径。如果未定义且启用了 `Leaf_CertificateCRL`，则会尝试自动下载 CRL。

WebAuthn 授权

如果您希望服务器在成功身份验证后发送一个可用于打开新 WebSocket 或 HTTP 连接的 Bearer 令牌，请传递参数 `token = true`。示例：

```
{
  "username": "alice@example.com", "token": true
}
```

成功认证后，服务器将发送如下响应：

```
{
  "verified": "ok",
  "authentication": {
    "token": "C760C1C39E3D4E829693A13F18F5CFDE537B516336FC48F7BAB0276176F9E6DE"
  }
}
```

当请求未获授权且连接将断开时，调用事件 **OnWebAuthnUnauthorized**。在此处可配置哪些端点需要 WebAuthn 身份验证，哪些不需要。

WebAuthn 授权 | HTTP

获得新令牌后，只需发送带有此 Bearer 令牌的授权头。您可以使用 TsgcHTTP1Client 的 CustomHeaders 属性配置 Bearer 令牌。示例：

```
String GetHTTPRequest(const String aURL, const String aToken)
{
    // Create the HTTP client dynamically
    std::unique_ptr<TsgcHTTP1Client> oHTTP(new TsgcHTTP1Client(nullptr));
    // Add Authorization header
    oHTTP->Request->CustomHeaders->AddValue("Authorization", "Bearer " + aToken);
    // Perform GET request and return result
    return oHTTP->Get(aURL);
}
```

WebAuthn 授权 | WebSocket

获得新令牌后，只需发送带有此 Bearer 令牌的授权头。您可以使用 OnHandShake 事件将 Bearer 令牌添加到连接请求中。示例：

```
void __fastcall OnClientHandshake(TsgcWSConnection *Connection, TStringList *Headers)
{
    Headers->Add("Authorization: Bearer C760C1C39E3D4E829693A13F18F5CFDE537B516336FC48F7BAB0276176F9E6DE");
}
```

WebAuthn | Javascript 客户端

WebAuthn (Web 身份验证 API) 是一项 W3C 标准，它使用**公钥密码学**实现安全的无密码身份验证。用户无需密码，而是使用**基于硬件的身份验证器**（如指纹读取器、Face ID、YubiKeys 等）或平台身份验证器（内置的，如 Touch ID）进行注册和身份验证。

以下展示如何使用 JavaScript 客户端处理注册和身份验证。

WebAuthn 注册

WebAuthn 注册流程

- **用户发起注册：**
 - 用户提供用户名并点击**注册**。
- **浏览器向服务器请求选项：**
 - 前端发起 POST 请求以获取注册选项。
- **浏览器创建凭据：**
 - 通过 `SimpleWebAuthnBrowser.startRegistration()` 使用 `navigator.credentials.create()` API 创建凭据。
- **服务器验证注册：**
 - 浏览器将凭据发送回服务器。
 - 服务器验证注册并为该用户存储公钥。

TsgcWSAPIServer_WebAuthn 组件包含一个用于测试 WebAuthn 协议的 HTML 文件。该 HTML 文件包含与 WebAuthn 交互所需的最小 UI 和 JavaScript。

sgcWebAuthn.html 演练

结构概览：

- **用户名输入：**捕获用户的标识符。
- **按钮：**
 - Register – 启动 WebAuthn 注册。
 - Authenticate – 启动登录（处理方式类似）。
- **调试控制台：**显示实时调试信息（来自 WebAuthn 的 JSON）。

1. 输入的 HTML UI

```
<input type="text" id="username" name="username" autocomplete="username webauthn" />
<button id="btnRegBegin"><strong>Register</strong></button>
```

2. JavaScript：按钮点击处理程序

```
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. 服务器响应（HTML 中的虚拟端点）

```
fetch('/sgcWebAuthn/Registration/Options', ...)
fetch('/sgcWebAuthn/Registration/Verify', ...)
```

4. 完成注册

```
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 身份验证

WebAuthn 身份验证工作原理

- **用户发起身份验证：**
 - 用户提供用户名并点击注册。
- **浏览器向服务器请求选项：**
 - 前端发出 POST 请求以获取身份验证选项。
- **浏览器创建凭据：**
 - 通过 `SimpleWebAuthnBrowser.startAuthentication()` 使用 `navigator.credentials.get()` API。
- **服务器验证身份：**
 - 浏览器将凭据发回给服务器。
 - 服务器验证已签名的结果。

1. HTML UI 设置

```
<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. 获取身份验证选项

在调用 `startAuthentication` 之前，您需要向服务器发送用户名

```
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'
  }),
});
```

服务器以包含以下内容的 JSON 对象响应：

```
{
  "challenge": "base64url-encoded-random-string",
  "allowCredentials": [
    {
      "id": "base64url-credential-id",
      "type": "public-key"
    }
  ],
  "userVerification": "preferred",
  "rpId": "yourdomain.com"
}
```

这称为 `PublicKeyCredentialRequestOptions`。

3. 接收验证器响应

`asseResp` 的简化示例如下：

```
{
  "id": "credentialId",
  "rawId": "base64url-encoded-id",
  "response": {
    "authenticatorData": "...",
    "clientDataJSON": "...",
    "signature": "...",
    "userHandle": "..."
  },
  "type": "public-key",
  "clientExtensionResults": {}
}
```

此响应证明用户：

- 拥有存储在验证器上的私钥
- 已签署服务器的质询
- 已实际到场（如有要求）

4. 向服务器发送签名的身份验证响应

用户通过身份验证器交互（通过 `startAuthentication()`）后，您将在 JavaScript 中获得一个响应对象。

```
const verificationResp = await fetch('/webauthn/authenticate/verify', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(asseResp), // this is the signed response
});
```

5. 获取服务器响应

服务器将以如下结果回复：

```
{ "verified": true }
```

或者，如果出现错误：

```
{ "verified": false, "error": "Invalid signature" }
```

您可以在前端代码中处理它：

```
const result = await verificationResp.json();
if (result.verified) {
  document.getElementById('Success').textContent = 'User authenticated!';
}
```



```
} else {  
  document.getElementById('Error').textContent = 'Authentication failed!';  
}
```

Amazon AWS | SQS

什么是 Amazon SQS？

Amazon Simple Queue Service (SQS) 是一项完全托管的消息队列服务，使您能够解耦和扩展微服务、分布式系统和无服务器应用程序。SQS 消除了管理和运营面向消息的中间件的复杂性和开销，使开发人员能够专注于差异化工作。使用 SQS，您可以在任意规模下在软件组件之间发送、存储和接收消息，而不会丢失消息，也无需其他服务保持可用。

优势

消除管理开销

使用 SQS，无需前期成本，无需购买、安装和配置消息传递软件，也无需耗时的基础设施构建和维护。

可靠地传递消息

SQS 允许您解耦应用程序组件，使它们可以独立运行和故障，从而提高系统的整体容错性。

确保敏感数据安全

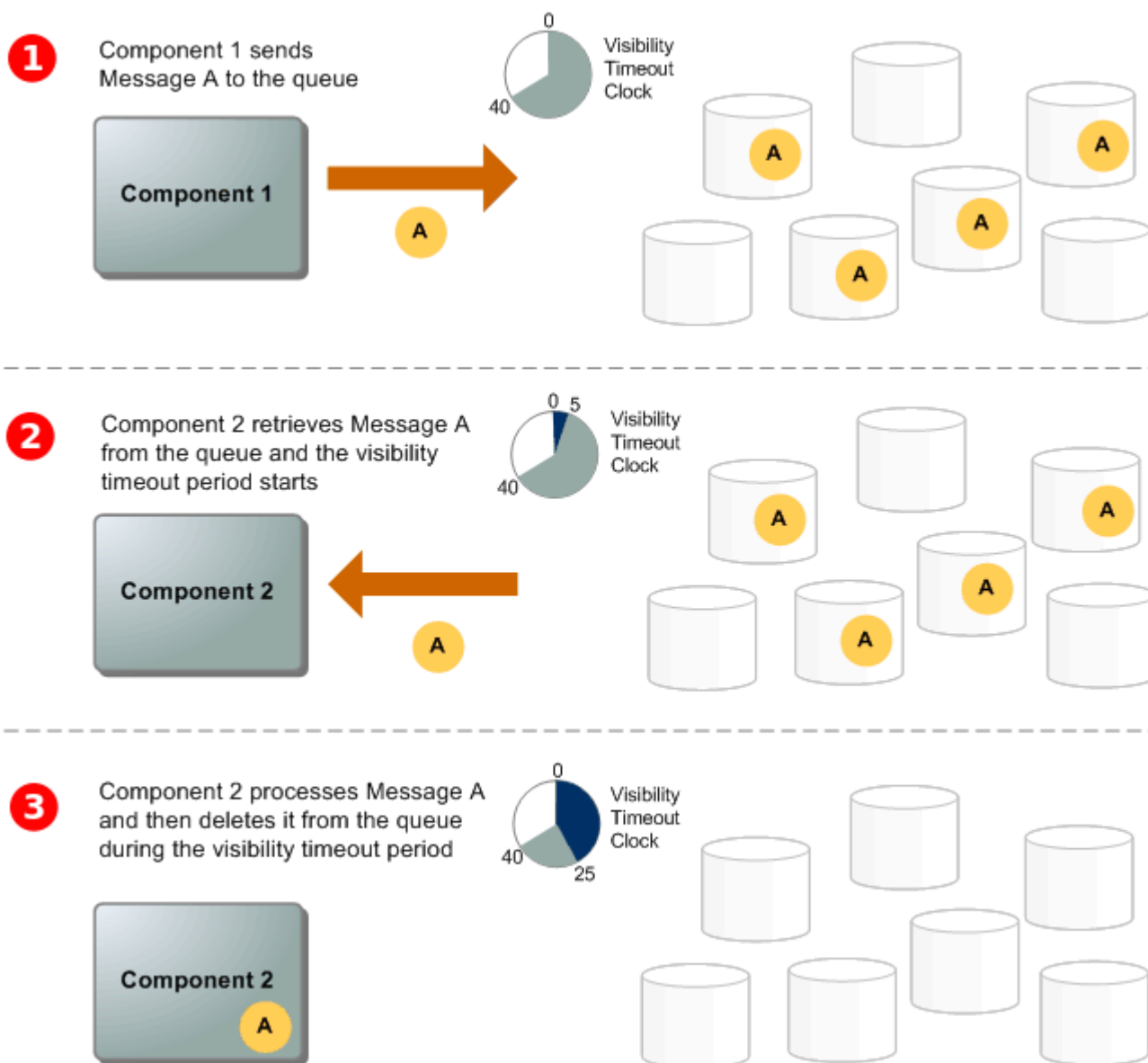
您可以使用 Amazon SQS 通过服务器端加密 (SSE) 对每条消息正文进行加密，从而在应用程序之间安全地交换敏感数据。

弹性扩展，成本效益高

SQS 可随应用程序弹性扩展，无需担心容量规划和预置问题。

工作流

以下场景描述了 Amazon SQS 消息在队列中从创建到删除的生命周期。



Amazon SQS 入门

在开始之前，请完成设置 Amazon SQS 的步骤。

步骤 1：创建队列

1. 登录到 Amazon SQS 控制台。
2. 选择“创建新队列”。
3. 在创建新队列页面上，确保您处于正确的区域，然后输入队列名称。
4. 默认选择 Standard。请选择 FIFO。
5. 要使用默认参数创建队列，请选择“快速创建队列”。

您的新队列已创建并在队列列表中被选中。

步骤 2：发送消息

创建队列后，您可以向其发送消息。以下示例展示了如何向现有队列发送消息。

1. 从队列列表中，选择您创建的队列。
2. 在队列操作中，选择“发送消息”。
3. 您的消息已发送，并显示“发送消息到 QueueName”对话框，其中展示了已发送消息的属性。

步骤 3：接收并删除您的消息

将消息发送到队列后，您可以消费它（从队列中检索）。从队列请求消息时，您无法指定要获取哪条消息。相反，您可以指定要获取的最大消息数量（最多 10 条）。

步骤 4：删除您的队列

如果您不使用 Amazon SQS 队列（且近期也不打算使用），最佳做法是将其从 Amazon SQS 中删除。

SQS 客户端

```
// TsgcHTTPAWS_SQS_Client is the component used for connect to Amazon SQS.
// Client connects using HTTPs protocol and authenticates using Access Key provided by Amazon.

// Before you attempt to connect to SQS service, you must set some data in AWSOptions property.

// Region: your endpoint region, example: us-east-1.
// AccessKey: access key provided by Amazon.
// SecretKey: secret key provided by Amazon.

// The following methods are supported by SQS client:

// AddPermission
// Adds a permission to a queue for a specific principal. This allows sharing access to the queue.

// ChangeMessageVisibility
// Changes the visibility timeout of a specified message in a queue to a new value. The default visibility
// The minimum is 0 seconds. The maximum is 12 hours.

// ChangeMessageVisibilityBatch
// Changes the visibility timeout of multiple messages. This is a batch version of ChangeMessageVisibility.
// The result of the action on each message is reported individually in the response. You can send up to 10

// CreateQueue
// Creates a new standard or FIFO queue. You can pass one or more attributes in the request.

vURL = SQS->CreateQueue("sqs_queue");
if (vURL != "")
{
    DoLog("#CreateQueue: " + vURL);
}

// DeleteMessage
// Deletes the specified message from the specified queue. To select the message to delete, use the Receipt

if (SQS->DeleteMessage("sqs_queue", "...receipt handle goes here...") == true)
{
    DoLog("#DeleteMessage: ok");
}
else
{
    DoLog("#DeleteMessage: error");
}

// DeleteMessageBatch
// Deletes up to ten messages from the specified queue. This is a batch version of DeleteMessage. The result

// DeleteQueue
// Deletes the queue specified by the queue name, regardless of the queue's contents.

if (SQS->DeleteQueue(txtQueueName->Text) == true)
{
    DoLog("#Delete Queue: ok");
}
else
{
    DoLog("#Delete Queue: error");
}

// GetQueueAttributes
// Gets attributes for the specified queue.
```

```

TsgcSQSAttributes *oAttributes = new TsgcSQSAttributes();
try
{
    if (SQS->GetQueueAttributes("sqs_queue", oAttributes) == true)
    {
        for (int i = 0; i < oAttributes->Count; i++)
        {
            DoLog("#Attribute: " + static_cast<TsgcSQSAttribute*>(oAttributes->Item[i])
                ->AttributeName + " " + static_cast<TsgcSQSAttribute*>(oAttributes->Item[i])
                ->AttributeValue);
        }
    }
    else
    {
        DoLog("#GetQueueAttributes: error");
    }
}
finally
{
    oAttributes->Free();
}

// GetQueueUrl
// Returns the URL of an existing Amazon SQS queue.

// ListDeadLetterSourceQueues
// Returns a list of your queues that have the RedrivePolicy queue attribute configured with a dead-letter

// ListQueueTags
// List all cost allocation tags added to the specified Amazon SQS queue.

// PurgeQueue
// Deletes the messages in a queue specified by the QueueName parameter.

if (SQS->PurgeQueue("sqs_queue") == true)
{
    DoLog("#PurgeQueue: ok");
}
else
{
    DoLog("#PurgeQueue: error");
}

// ReceiveMessage
// Retrieves one or more messages (up to 10), from the specified queue.

TsgcSQSReceiveMessageResponses *oResponses = new TsgcSQSReceiveMessageResponses();
try
{
    if (SQS->ReceiveMessage("sqs_test", oResponses) == true)
    {
        for (int i = 0; i < oResponses->Count; i++)
        {
            DoLog("#ReceiveMessage: " + static_cast<TsgcSQSReceiveMessageResponse*>(oResponses->Item[i])->Body)
                FReceiptHandle = static_cast<TsgcSQSReceiveMessageResponse*>(oResponses->Item[i])->ReceiptHandle;
        }
    }
}
finally
{
    oResponses->Free();
}

// RemovePermission
// Revokes any permissions in the queue policy that matches the specified Label parameter.

// SendMessage
// Delivers a message to the specified queue.

if (SQS->SendMessage("sqs_queue", "My First Message") == true)
{
    DoLog("#SendMessage: ok");
}
else
{
    DoLog("#SendMessage: error");
}

// SendMessageBatch
// Delivers up to ten messages to the specified queue. This is a batch version of SendMessage.

// SetQueueAttributes
// Sets the value of one or more queue attributes. When you change a queue's attributes, the change can tak
// for most of the attributes to propagate throughout the Amazon SQS system.

```

```

TsgcSQSAttributes *oAttributes = new TsgcSQSAttributes();
try
{
    oAttributes->AddSQSAttribute(sqsatVisibilityTimeout, "45");
    if (SQS->SetQueueAttributes("sqs_queue", oAttributes) == true)
    {
        DoLog("#SetQueueAttributes: ok");
    }
    else
    {
        DoLog("#SetQueueAttributes: error");
    }
}
finally
{
    oAttributes->Free();
}

// TagQueue
// Add cost allocation tags to the specified Amazon SQS queue.

// UntagQueue
// Remove cost allocation tags from the specified Amazon SQS queue.

```

事件

OnSQSBeforeRequest

此事件在 sqs 组件发起 HTTP 请求之前调用。您可以访问 URL 参数，如果 Handled 参数设为 True，则表示组件不会发起 HTTP 请求。

OnSQSError

如果组件在发起请求时出现任何错误，将调用此事件，并提供错误代码和错误描述。

OnSQSResponse

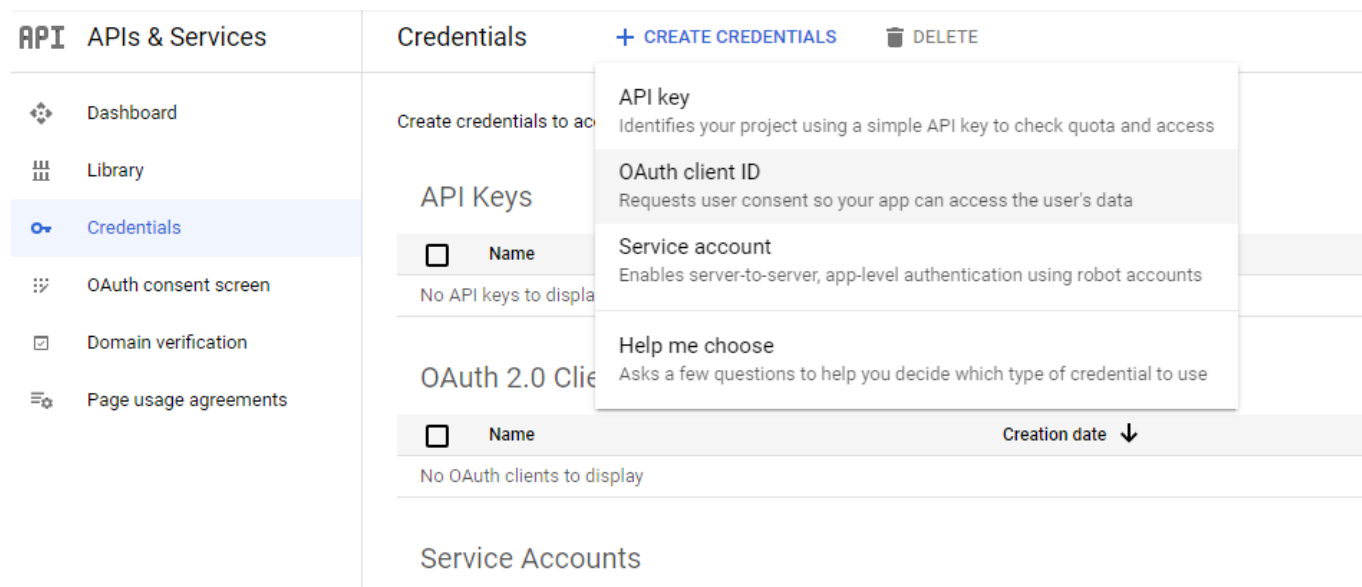
此事件在处理带有服务器原始响应的 HTTP 请求后调用。

Google Cloud | Google OAuth2 密钥

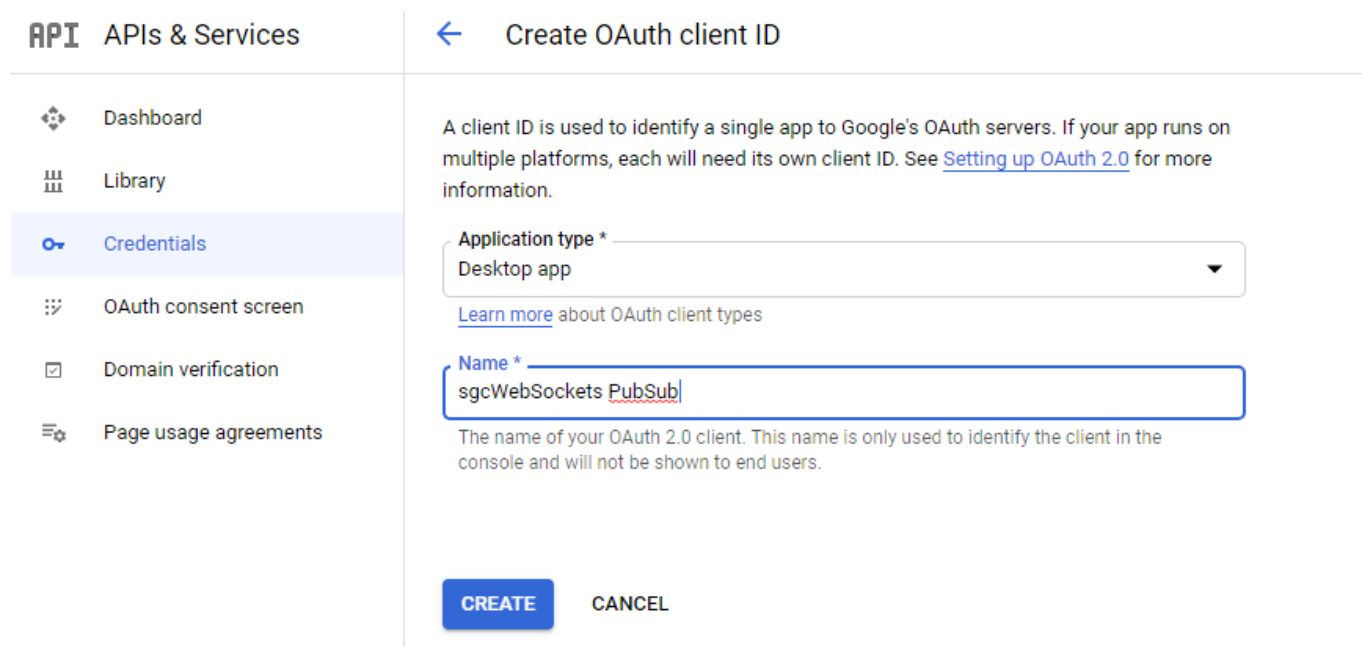
要使用 sgcWebSockets Google Cloud 组件并通过 OAuth2 进行身份验证，首先必须从 Google Cloud 获取 OAuth2 密钥。

请查阅以下步骤，了解如何获取 Google OAuth2 密钥及如何在我们的 PubSub 示例应用中配置它们。

首先登录您的 **Google Cloud** 账户，然后使用现有项目或创建一个新项目。完成后，前往凭据菜单，点击 **创建凭据** 按钮，选择 **OAuth 客户端 ID** 选项。



选择您的应用程序类型并设置一个描述性名称。



若操作成功，您将获得客户端 ID 和客户端密钥。

OAuth client created

The client ID and secret can always be accessed from Credentials in APIs & Services



OAuth is limited to 100 [sensitive scope logins](#) until the [OAuth consent screen](#) is verified. This may require a verification process that can take several days.

Your Client ID

843483347040-ehcskpfsp4180r1bdfoe6mc32e3ncmn0.apps.gc



Your Client Secret

pvogD9reE0t9ilL6eR1jE60Z



OK

请勿与任何人共享您的 OAuth2 数据！

现在将其复制到 sgcWebSockets PubSub 示例中，并添加项目 ID（不是项目名称）

Select a project

 NEW PROJECT

 Search projects and folders

RECENTALL

	Name	ID
 	PubSub 	pubsub-270909
	GmailApiTest 	gmailapitest-265010
	My Project 	melodic-voice-265009

CANCEL

OPEN

这是必须在 `sgcWebSocket PubSub` 示例中配置的方式。

Google Pub/Sub API Client

Google API

Client Id843483347040-ehcskpfsp4180rlbdfoe6mc32e3ncmn0.app

Client SecretpvogD9reEOt9iLL6eR1jE60Z

Publisher

Project-idpubsub-270909Topictopic-1

Create TopicDelete TopicGet TopicList Topics

PublishMessage sent from sgcWebSockets Pub/Sub API.

Subscriber

Project-idSubscriptionsubscription-1Topictopic-1

Create Subs.Delete Subs.Get Subs.List Subs.

PullAcknowledge

然后您可以尝试创建一个新主题（例如）。第一次时，您必须授权 OAuth2 连接，因此将显示一个新的 Web 浏览器，请求使用 Google 提供的 OAuth2 凭据授权访问您的账户。

Sign in with Google

Confirm your choices

 sgomez@gmail.com

You are allowing PubSub to:

☒ View and manage Pub/Sub topics and subscriptions

Make sure you trust PubSub

You may be sharing sensitive info with this site or app.
Learn about how PubSub will handle your data by reviewing
its terms of service and privacy policies. You can always
see or remove access in your [Google Account](#).

[Learn about the risks](#)

Cancel Allow

允许连接，成功后即可开始使用此 API。

Google Pub/Sub API Client

Google API

Client Id

>4180rlbdf6e6mc32e3ncmn0.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 服务账号

要使用 sgcWebSockets Google Cloud 组件并通过服务账号进行身份验证，首先需要获取私钥证书。来自 Google Cloud。
请参阅以下步骤以获取 Google 私钥证书，以及如何在我们的 PubSub 示例应用程序中进行配置。

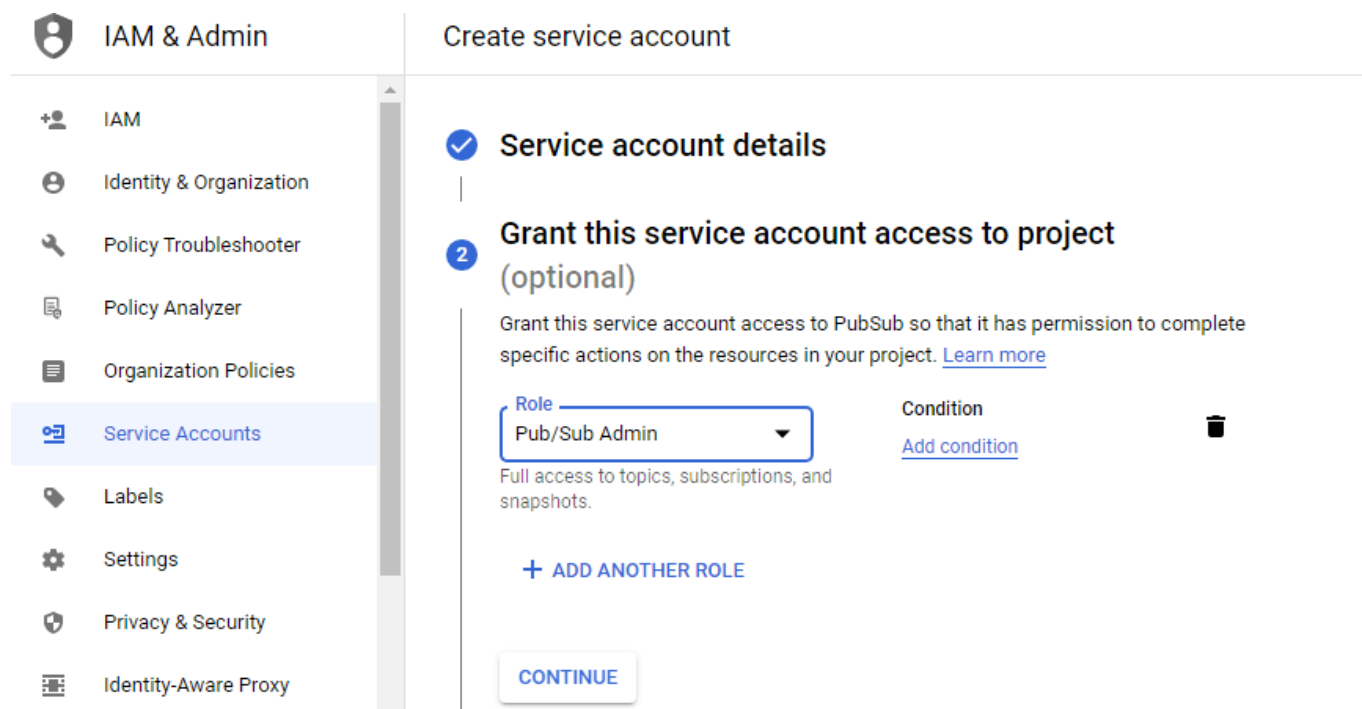
首先登录您的 **Google Cloud 账户**，然后使用现有项目或创建一个新项目。

The screenshot shows the 'IAM & Admin' console. The left sidebar has 'Service Accounts' selected. The main content area is titled 'Service accounts for project "PubSub"'. It includes a '+ CREATE SERVICE ACCOUNT' button and a 'DELETE' button. Below this, there is explanatory text about service accounts and a table with columns: Email, Status, Name, Description, Key ID, Key creation date, and Actions. The table currently shows 'No rows to display'.

选择 **CREATE SERVICE ACCOUNT**，将显示一个新页面，您必须在其中设置服务账户名称和描述

The screenshot shows the 'Create service account' page. The left sidebar has 'Service Accounts' selected. The main content area is titled 'Create service account' and contains a section '1 Service account details'. This section has three input fields: 'Service account name' (with the value 'sgcServiceAccount'), 'Service account ID' (with the value 'sgcserviceaccount' and a domain '@pubsub-270909.iam.gserviceaccount.com'), and 'Service account description' (with the value 'Service Account Test'). A 'CREATE' button is at the bottom.

然后至少选择一个角色。此处选择 PubSub Admin，以允许客户端向主题发布和订阅，但您也可以选择权限较少的其他角色。



IAM & Admin

- IAM
- Identity & Organization
- Policy Troubleshooter
- Policy Analyzer
- Organization Policies
- Service Accounts**
- Labels
- Settings
- Privacy & Security
- Identity-Aware Proxy

Create service account

- Service account details
- Grant this service account access to project (optional)**

Grant this service account access to PubSub so that it has permission to complete specific actions on the resources in your project. [Learn more](#)

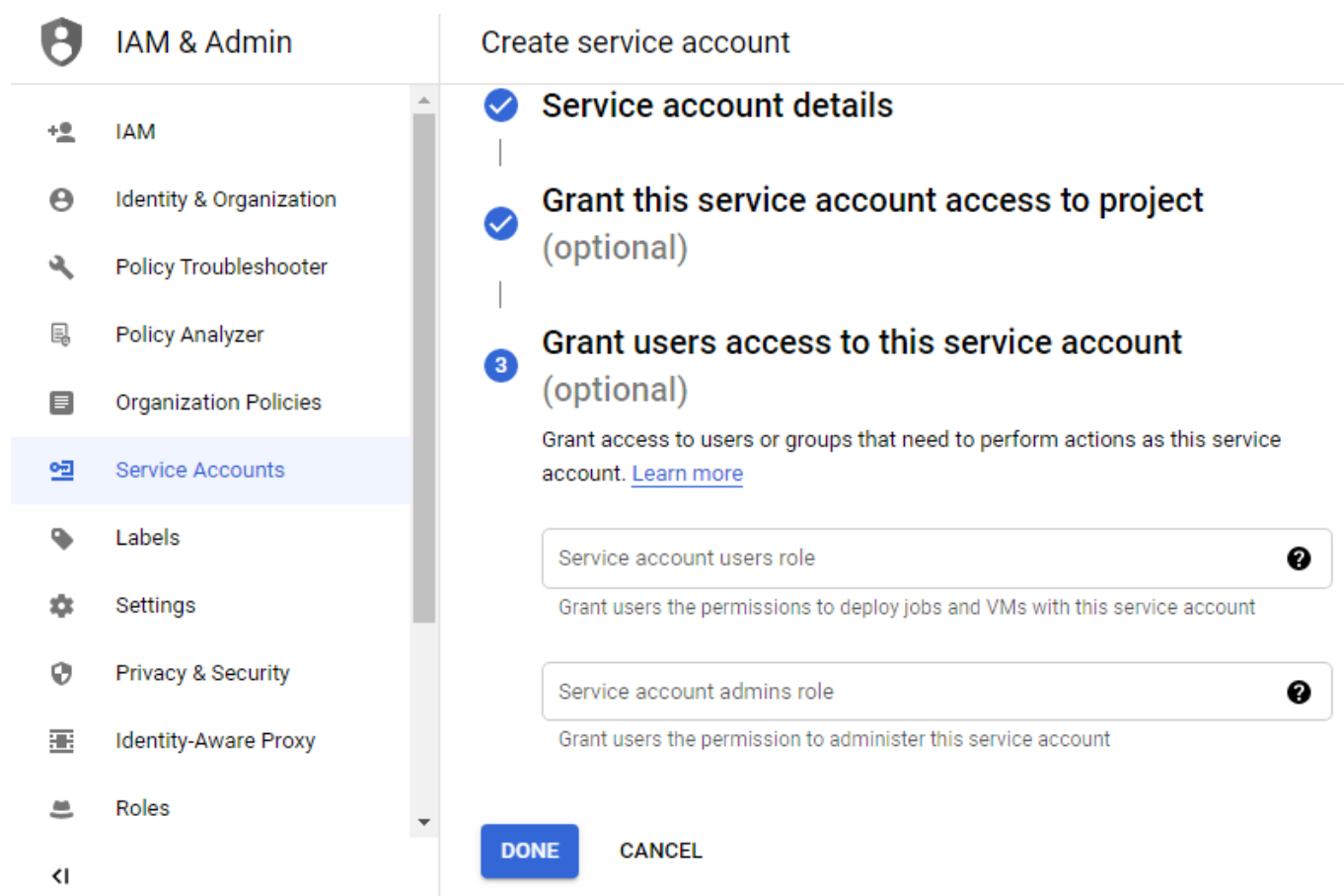
Role: **Pub/Sub Admin** Condition: [Add condition](#)

Full access to topics, subscriptions, and snapshots.

[+ ADD ANOTHER ROLE](#)

CONTINUE

按 CONTINUE 键，最终即可向其他用户授予访问权限



IAM & Admin

- IAM
- Identity & Organization
- Policy Troubleshooter
- Policy Analyzer
- Organization Policies
- Service Accounts**
- Labels
- Settings
- Privacy & Security
- Identity-Aware Proxy
- Roles

Create service account

- Service account details
- Grant this service account access to project (optional)
- Grant users access to this service account (optional)**

Grant access to users or groups that need to perform actions as this service account. [Learn more](#)

Service account users role ?
Grant users the permissions to deploy jobs and VMs with this service account

Service account admins role ?
Grant users the permission to administer this service account

DONE CANCEL

完成后按"完成"，将显示一条新记录。

IAM & Admin
IAM
Identity & Organization
Policy Troubleshooter
Policy Analyzer
Organization Policies
Service Accounts
Labels

Service accounts
+ CREATE SERVICE ACCOUNT
DELETE

Service accounts for project "PubSub"

A service account represents a Google Cloud service identity, such as code running on Compute Engine VMs, App Engine apps, or system [about service accounts](#).

Organization policies can be used to secure service accounts and block risky service account features, such as automatic IAM Grants, ke service accounts entirely. [Learn more about service account organization policies](#).

Filter table

☐	Email	Status	Name ↑	Description	Key ID
☐	 sgcserviceaccount@pubsub-270909.iam.gserviceaccount.com	✓	sgcServiceAccount	Service Account Test	No keys

下一步是创建新密钥，因此在操作列中选择"创建密钥"选项。选择 JSON 以 JSON 格式下载配置，新密钥将被创建。

Service accounts for project "PubSub"

A service account represents a Google Cloud service identity, such as code running on Compute Engine VMs, App Engine apps, or systems running outside Google. [Learn more about service accounts](#).

Organization policies can be used to secure service accounts and block risky service account features, such as automatic IAM Grants, key creation/upload, or the creation of service accounts entirely. [Learn more about service account organization policies](#).

Filter table							
☐	Email	Status	Name ↑	Description	Key ID	Key creation date	Actions
☐	 sgcserviceaccount@pubsub-270909.iam.gserviceaccount.com	✓	sgcServiceAccount	Service Account Test	425a876f68b8b2a66f2fcc5b2dee8e918b0eb9ab	Dec 20, 2020	⋮

最后，您只需填写 Google 在 sgcsWebSockets PubSub 客户端中提供的数据。您可以使用 `LoadSettingsFromFile` 加载配置 JSON 文件。

Google Pub/Sub API Client

OAuth2 Service Account

Client Email: sgcserviceaccount@pubsub-270909.iam.gserviceaccount.com

Private Key Id: 425a876f68b8b2a66f2fcc5b2dee8e918b0eb9ab

Private Key: -----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBCwggSjAgEAAoIBA
QC3xz0L2Wkr9Hvj
HOMMf5Qum78NI1c8nVcF8HlQVYr8amSLV

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

什么是 Google Cloud Pub/Sub ?

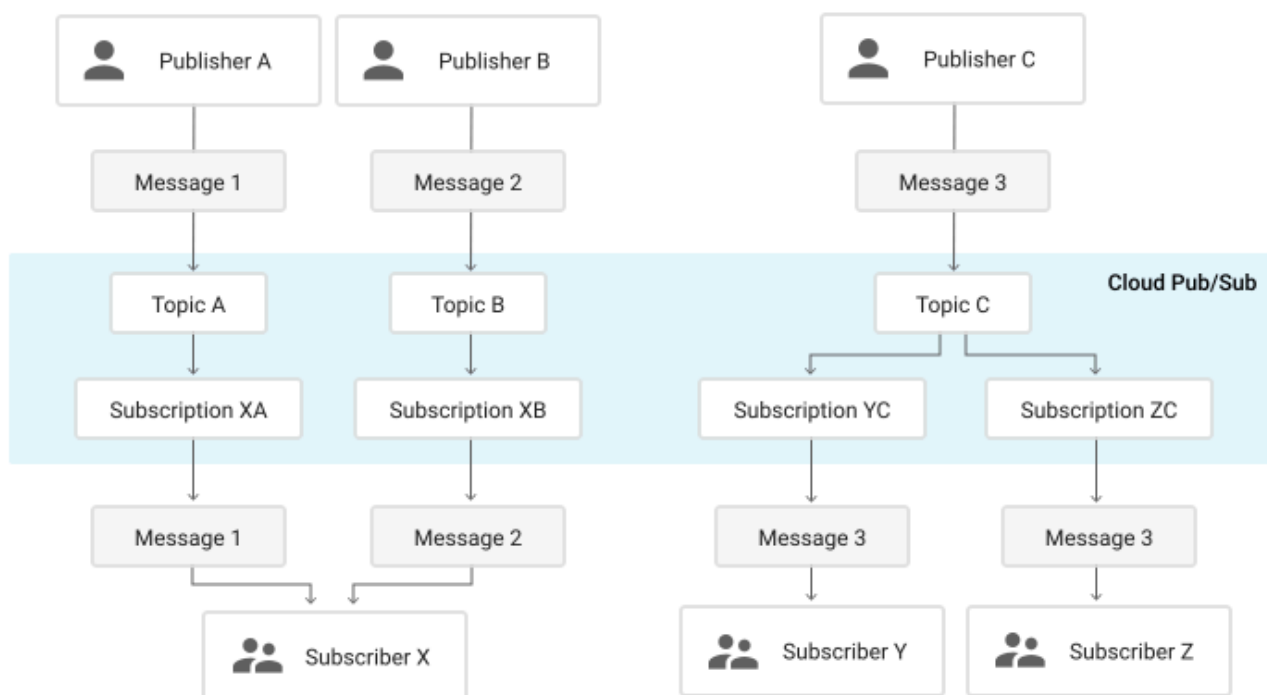
Pub/Sub 将企业消息中间件的灵活性和可靠性引入云端。同时，Pub/Sub 是一个可扩展、持久的事件摄取和传递系统，作为现代流分析管道的基础。通过提供多对多的异步消息传递，将发送者和接收者解耦，它实现了独立编写的应用程序之间的安全且高度可用的通信。Pub/Sub 提供低延迟、持久的消息传递，帮助开发者快速集成托管在 Google Cloud Platform 上和外部的系统。

功能特性

至少一次投递 同步的跨区域消息复制和每消息接收跟踪，确保在任何规模下都能实现至少一次投递。	开放 七种语言的开放 API 和客户端库，支持跨云和混合部署。	精确一次处理 Cloud Dataflow 支持对 Cloud Pub/Sub 流进行可靠、富有表达力的精确一次处理。
默认全球可用 从世界任何地方发布，从任何地方消费，延迟一致。无需复制。	无需配置，一切自动 Cloud Pub/Sub 没有分片或分区。只需设置配额，发布，然后消费。	合规与安全 Cloud Pub/Sub 是符合 HIPAA 的服务，提供细粒度的访问控制和端到端加密。
集成 充分利用与多种服务的集成，例如 Cloud Storage 和 Gmail 更新事件，以及用于无服务器事件驱动计算的 Cloud Functions。	查找和重放 将您的积压消息倒回到任意时间点或快照，从而可以重新处理消息。快进以丢弃过时数据。	

发布者-订阅者关系

发布者应用程序创建消息并将其发送到主题。订阅者应用程序创建对主题的订阅以接收消息。通信可以是一对多（扇出）、多对一（扇入）或多对多。



常见用例

- **在网络集群中均衡工作负载。**例如，大量任务队列可以高效分配给多个工作节点，例如 Google Compute Engine 实例。
- **实现异步工作流。**例如，订单处理应用程序可以将订单放置到某个主题上，由一个或多个工作者对其进行处理。
- **分发事件通知。**例如，接受用户注册的服务可以在每次新用户注册时发送通知，下游服务可以订阅以接收事件通知。
- **刷新分布式缓存。**例如，应用程序可以发布失效事件来更新已更改对象的 ID。
- **记录到多个系统。**例如，Google Compute Engine 实例可以将日志写入监控系统、写入数据库以供后续查询等。
- **来自各种流程或设备的数据流。**例如，住宅传感器可以将数据流式传输到云中托管的后端服务器。
- **可靠性提升。**例如，单可用区的 Compute Engine 服务可通过订阅公共主题，在其他可用区运行，以从可用区或区域故障中恢复。

授权

Google Pub/Sub 组件客户端可以使用以下方法登录 Google 服务器：

- **gcaOAuth2**：OAuth2 协议
- **gcaJWT**：JWT 令牌。

OAuth2

登录通过网页浏览器完成，用户使用自己的账户登录并授权 PubSub 请求。

- **GoogleCloudOptions.OAuth2.ClientId**：是 Google 为通过 OAuth2 协议进行身份验证所提供的 ClientID。
- **GoogleCloudOptions.OAuth2.ClientSecret**：Google 提供的客户端密钥字符串，用于通过 OAuth2 协议进行身份验证。
- **GoogleCloudOptions.OAuth2.Scope**：是 OAuth2 的范围，通常无需修改默认值，除非需要获取超出默认范围的访问权限。
- **GoogleCloudOptions.OAuth2.LocalIP**：OAuth2 协议需要一个本地服务器来监听来自身份验证服务器的响应。这是 IP 或 DNS 名称。默认情况下为 127.0.0.1。

- **GoogleCloudOptions.OAuth2.LocalPort**：本地服务器监听端口。
- **GoogleCloudOptions.OAuth2.RedirectURL**：如果您需要设置与 LocalPort + LocalIP 不同的重定向 URL，可以在此属性中设置（示例：<http://127.0.0.1:8080/oauth2>）。

服务账号

登录通过使用 Google 提供的私钥对请求进行签名来完成。此方法推荐用于无需用户交互的自动化服务或应用程序。

- **GoogleCloudOptions.JWT.ClientEmail**：创建新服务账户时提供的客户端邮箱名称。即 JSON 配置文件中的 "client_email" 节点。
- **GoogleCloudOptions.JWT.PrivateKeyId**：Google 提供的私钥 ID。JSON 配置文件中的 "private_key_id" 节点。
- **GoogleCloudOptions.JWT.PrivateKey**：Google 提供的私钥证书。JSON 配置文件中的 "private_key" 节点。

TLS 选项

TLSOptions 集中管理用于与 Google Cloud Pub/Sub 通信的安全通道配置。

- **TLSOptions.IOHandler**：选择适合您部署环境的 TLS 实现（OpenSSL、SChannel 等）。
- **TLSOptions.Version**：当您的基础设施有要求时，限制协商使用特定 TLS 版本。
- **TLSOptions.VerifyCertificate**：控制证书验证，启用更严格的安全策略。
- **TLSOptions.OpenSSL_Options.LibPath**：设置包含随应用程序一起发布的 OpenSSL 二进制文件的目录。
- **TLSOptions.SChannel_Options.UseLegacyCredentials**：在 SChannel 需要时激活 Windows 旧版凭据处理。

示例：

```
TsgcHTTPGoogleCloud_PubSub_Client *oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client(NULL);
oPubSub->TLSOptions->IOHandler = iohOpenSSL;
oPubSub->TLSOptions->Version = tls1_3;
oPubSub->TLSOptions->VerifyCertificate = true;
oPubSub->TLSOptions->OpenSSL_Options->LibPath = oslpDefaultFolder;
```

创建新服务账户时，可以下载包含所有配置的 JSON 文件。PubSub 组件可以处理此文件，只需调用方法 **LoadSettingsFromFile** 并将 JSON 文件名作为参数传入即可。

最常见的用途

- 配置
 - [Google OAuth2 密钥](#)
 - [服务账号](#)

Google Pub/Sub 客户端

OAuth2

要使用 Google Pub/Sub API，sgcWebSockets Pub/Sub 组件默认使用 OAuth2 作为认证方式，因此首先必须从您的 Google 账户设置 **ClientId** 和 **ClientSecret**。

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->Authorization = gcaOAuth2;
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
```

服务账号

服务账号需要构建 JWT 并将其作为授权令牌传递。

```
TsgcHTTPGoogleCloud_PubSub_Client *oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client(this);
oPubSub->GoogleCloudOptions->Authorization = gcaJWT;
oPubSub->GoogleCloudOptions->JWT->ClientEmail = "...google email...";
oPubSub->GoogleCloudOptions->JWT->PrivateKeyId = "...private key id...";
oPubSub->GoogleCloudOptions->JWT->PrivateKey->Lines->Text = "...private key certificate...";
```

这是从 Google 获取授权令牌密钥所必需的，该密钥将用于所有 REST API 调用。

所有方法都返回响应，该响应可能成功或返回错误。

Projects.Snapshots

方法	参数	描述	示例
CreateSnapshot	project, snapshot, subscription	从所请求的订阅创建快照。快照用于 subscriptions.seek 操作，该操作允许您批量管理消息确认状态，即将现有订阅中消息的确认状态设置为快照所捕获的状态。	CreateSnapshot('pubsub-270909', 'snapshot-1', 'subscription-1')
DeleteSnapshot	project, snapshot	删除现有快照	DeleteSnapshot('pubsub-270909', 'snapshot-1')
ListSnapshots	project	列出现有快照	ListSnapshots('pubsub-270909')

Projects.Subscriptions

方法	参数	描述	示例
AcknowledgeSubscription			
CreateSubscription	project, subscription, topic	创建对指定主题的订阅。如果订阅已存在，则返回 ALREADY_EXISTS。如果对应主题不存在，则返回 NOT_FOUND。	CreateSubscription('pubsub-270909', 'subscription-1', 'topic-1')
DeleteSubscription	project, subscription	删除现有订阅。订阅中保留的所有消息将立即丢弃。	DeleteSubscription('pubsub-270909', 'subscription-1')
GetSubscription	project, subscription	获取订阅的配置详情。	GetSubscription('pubsub-270909', 'subscription-1')
ListSubscriptions	project	列出匹配的订阅。	ListSubscriptions('pubsub-270909', 'subscription-1')
ModifyAckDeadlineSubscription	project, subscription, AckIds	修改特定消息的确认截止时间。此方法适用于表示订阅者需要更多时间处理消息，或在处理中断时将消息重新发	

		布。请注意，这不会修改用于后续消息的订阅级 ackDeadlineSeconds。	
ModifyPushConfigSubscription	project, subscription	修改指定订阅的 PushConfig。可用于将推送订阅更改为拉取订阅（通过传入空 PushConfig 表示），或反向操作，也可更改推送订阅的端点 URL 及其他属性。无论 PushConfig 如何变更，消息将在整个调用过程中持续累积并等待投递。	
Pull	project, subscription	从服务器拉取消息。如果给定订阅的并发拉取请求过多，服务器可能返回 UNAVAILABLE。	<code>pull('pubsub-270909', 'subscription-1')</code>
Seek	project, subscription, timeUTC, snapshot	将现有订阅定位到特定时间点或给定快照（以请求中提供的为准）。快照用于 subscriptions.seek 操作，可批量管理消息确认状态，即将现有订阅中消息的确认状态设置为快照所捕获的状态。注意，订阅和快照必须位于同一主题下。	

Projects.Topics

方法	参数	描述	示例
CreateTopic	project, topic	创建具有给定名称的主题	<code>CreateTopic('pubsub-270909', 'topic-1')</code>
DeleteTopic	project, topic	删除具有给定名称的主题。如果主题不存在则返回 NOT_FOUND。删除主题后，可以使用相同名称创建新主题，但这是一个全新的主题，没有任何旧配置或订阅。	<code>DeleteTopic('pubsub-270909', 'topic-1')</code>
GetTopic	project, topic	获取主题的配置信息。	<code>GetTopic('pubsub-270909', 'topic-1')</code>
ListTopics	project	列出匹配的主题。	<code>ListTopics('pubsub-270909')</code>

发布	project, topic, message	向主题添加一条或多条消息。若主题不存在, 返回 NOT_FOUND。	Publish('pubsub-270909', 'topic-1', 'My First PubSub Message.')

Projects.Topics.Subscriptions

方法	参数	描述	示例
ListTopicSubscriptions	project, topic	列出此主题上的订阅名称。	ListTopicSubscriptions('pubsub-270909', 'topic-1')

最常用的方法

以下是与 Google Cloud Pub/Sub API 配合使用的最常用方法。

如何创建新主题

为 ID 为 pubsub-270909 的项目创建名称为 topic-1 的新主题。

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
oPubSub->CreateTopic("pubsub-270909", "topic-1");
```

来自服务器的响应

```
{
  "name": "projects/pubsub-270909/topics/topic-1"
}
```

发布消息

在新创建的主题中发布新消息。

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
oPubSub->Publish("pubsub-270909", "topic-1", "My First Message from sgcWebSockets.");
```

来自服务器的响应

```
{
  "messageIds": [
    "1050732082561505"
  ]
}
```

```
}
}
```

发布带属性的消息

```
TsgcHTTPGoogleCloud_PubSub_Client *oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
oAttributes = new TStringList();
try
{
    oAttributes->CommaText = "origin=gcloud-sample,username=gcp";
    oPubSub->Publish("pubsub-270909", "topic-1", "My First Message from sgcWebSockets.", oAttributes, "username"));
}
__finally
{
    oAttributes->Free();
}
```

如何创建新订阅

为 id 为 pubsub-270909 的项目创建新订阅，订阅名称为 subscription-1，主题为 topic-1

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
oPubSub->CreateSubscription("pubsub-270909", "subscription-1", "topic-1");
```

来自服务器的响应

```
{
  "name": "projects/pubsub-270909/subscriptions/subscription-1",
  "topic": "projects/pubsub-270909/topics/topic-1",
  "pushConfig": {},
  "ackDeadlineSeconds": 10,
  "messageRetentionDuration": "604800s",
  "expirationPolicy": {
    "ttl": "2678400s"
  }
}
```

如何从订阅中读取消息

读取之前创建的订阅中的消息。

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oPubSub->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
oPubSub->pubsub->Pull("pubsub-270909", "subscription-1");
```

来自服务器的响应

```
{
  "receivedMessages": [
    {
      "ackId": "PjA-RVNEUAYWLF1GSFE3GQhouU5PXiM_NSAorREFC08CKF15MEorQVh0Dj4N",
      "message": {
        "data": "TXkgRmlyc3QgTWVzc2FnZSBmcm9tIHNNY1dlYlNvY2tldHMu",
        "messageId": "1050732082561505",
        "publishTime": "2020-03-14T15:25:31.505Z"
      }
    }
  ]
}
```

消息以 Base64 编码接收，因此必须先解码才能读取内容。

```
sgcBase_Helpers->DecodeBase64("TXkgRm1yc3QgTWVzc2FnZSBmcm9tIHNNY1d1Y1NvY2tldHMu=");
```


Google Cloud | 日历

Google Calendar API 允许您将应用程序与 Google 日历集成，为您创造与用户互动的新方式。Calendar API 允许您显示、创建和修改日历事件，以及处理许多其他与日历相关的对象，例如日历或访问控制。

API 资源

Google Calendar 使用以下资源：

- **事件**：日历中的一个事件，包含标题、开始和结束时间以及与会者等信息。事件可以是单次事件或重复事件。事件由 Event 资源表示。给定日历的 Events 集合包含该日历的所有事件资源。
- **日历**：日历是事件的集合。每个日历都有相关联的元数据，例如日历描述或默认日历时区。单个日历的元数据由 Calendar 资源表示。Calendars 集合包含所有现有日历的 Calendar 资源。
- **CalendarList**：用户在 Calendar UI 中的日历列表中所有日历的列表。日历列表中单个日历的元数据由 CalendarListEntry 资源表示。此元数据包括日历的用户特定属性，例如颜色或新事件的通知。CalendarList 集合包含给定用户的所有 CalendarListEntry 资源。有关 Calendars 和 CalendarList 集合之间差异的进一步说明，请参阅“日历和日历列表”。
- **Setting**：来自 Calendar UI 的用户偏好设置，例如用户的时区。单个用户偏好由一个 Setting 资源表示。Settings 集合包含指定用户的所有 Setting 资源。
- **ACL**：访问控制规则，向用户（或一组用户）授予对日历的指定访问级别。单条访问控制规则由一个 ACL 资源表示。给定日历的 ACL 集合包含授予该日历访问权限的所有 ACL 资源。
- **Color**：日历 UI 中显示的颜色。Colors 资源代表日历 UI 中所有可用颜色的集合，分为两组：可用于事件的颜色和可用于日历的颜色。
- **空闲/忙碌**：日历有已安排事件的时段视为“忙碌”，无事件的时段视为“空闲”。Freebusy 资源允许查询给定日历或一组日历的忙碌时段集合。

主要功能

- 功能完整的 Google 日历客户端 API V3。
- API 支持的所有方法均可使用客户端 API 调用。
- 客户端使用 HTTP/2 协议发送请求（*仅 Enterprise 版本）。
- 使用 PageNextToken 自动处理分页响应。
- 轻松访问日历和事件数据属性。
- 身份验证方法：
 - OAuth2：需要用户交互。
 - 服务账户（需要域范围委派）：适用于 Windows 服务、守护进程等。

配置

Google Calendar 组件客户端具有以下属性：

OAuth2

- **GoogleCloudOptions.OAuth2.ClientId**：是 Google 为通过 OAuth2 协议进行身份验证所提供的 ClientID。
- **GoogleCloudOptions.OAuth2.ClientSecret**：Google 提供的客户端密钥字符串，用于通过 OAuth2 协议进行身份验证。
- **GoogleCloudOptions.OAuth2.Scope**：是 OAuth2 的范围，通常无需修改默认值，除非需要获取超出默认范围的访问权限。
- **GoogleCloudOptions.OAuth2.LocalIP**：OAuth2 协议需要一个本地服务器来监听来自身份验证服务器的响应。这是 IP 或 DNS 名称。默认情况下为 127.0.0.1。
- **GoogleCloudOptions.OAuth2.LocalPort**：本地服务器监听端口。
- **GoogleCloudOptions.OAuth2.RedirectURL**：如果您需要设置与 LocalPort + LocalIP 不同的重定向 URL，可以在此属性中设置（示例：<http://127.0.0.1:8080/oauth2>）。

您可以使用 Scopes 属性修改客户端 API 的作用域，只需选择您的客户端支持的作用域即可。

JWT

登录是通过使用 Google 提供的私钥对请求进行签名来完成的，此方法推荐用于无需用户交互的自动化服务或应用程序。需要使用[域范围委派](#)配置服务账号。

- **GoogleCloudOptions.JWT.ClientEmail**：创建新服务账户时提供的客户端邮箱名称。即 JSON 配置文件中的 "client_email" 节点。
- **GoogleCloudOptions.JWT.PrivateKeyId**：Google 提供的私钥 ID。JSON 配置文件中的 "private_key_id" 节点。
- **GoogleCloudOptions.JWT.PrivateKey**：Google 提供的私钥证书。JSON 配置文件中的 "private_key" 节点。
- **GoogleCloudOptions.JWT.Subject**：这是通过域级委派与服务账号关联的工作区电子邮件账号。

TLS 选项

使用 **TLSOptions** 属性自定义与 Google 服务器建立的安全连接。

- **TLSOptions.IOHandler**：选择 TLS 协议栈（OpenSSL、SChannel 等）。
- **TLSOptions.Version**：强制使用特定的 TLS 协议版本（例如 tls1_2 或 tls1_3）。
- **TLSOptions.VerifyCertificate**：启用或禁用服务器证书验证。
- **TLSOptions.OpenSSL_Options.LibPath**：指向包含随应用程序部署的 OpenSSL 库的文件夹。
- **TLSOptions.SChannel_Options.UseLegacyCredentials**：启用某些 Windows 版本在使用 SChannel 时所需的旧版凭证流程。

以下代码段展示了如何配置 TLS 选项：

```
TsgcHTTPGoogleCloud_Calendar_Client *oCalendar = new TsgcHTTPGoogleCloud_Calendar_Client(NULL);
oCalendar->TLSOptions->IOHandler = iohOpenSSL;
oCalendar->TLSOptions->Version = tls1_3;
oCalendar->TLSOptions->VerifyCertificate = true;
oCalendar->TLSOptions->OpenSSL_Options->LibPath = opensslDefaultFolder;
```

最常见的用途

- **配置**
 - [Google Calendar 刷新令牌](#)
 - [Google Calendar 服务账号](#)
- **同步**
 - [Google Calendar 同步日历](#)
 - [Google Calendar 同步事件](#)

同步日历

TsgcHTTPGoogleCloud_Calendar_Client 组件允许您使用直接 Google API 调用或使用我们简易的 Calendars 方法来同步日历。

方法	参数	描述
NewCalendar	aSummary ：日历的标题。	创建新日历
DeleteCalendar	aid ：日历的标识符。	删除现有日历。
UpdateCalendar	aResource ：包含日历数据的对象。	更新现有日历。
LoadCalendars		加载所有日历，并用这些数据填充 Calendars 属性。

LoadCalendarsChanged	aSyncToken ：用于更新您日历的最后一个令牌。	从设置的 Token 加载日历的所有更改。

日历客户端有一个名为 **Calendars** 的属性，调用任何前述方法后，您可以通过该属性访问日历数据。此属性自动同步。

同步事件

TsgcHTTPGoogleCloud_Calendar_Client 组件允许您通过直接 Google API 调用或使用我们便捷的事件方法来同步日历事件。

方法	参数	描述
NewEvent	aCalendarId ：日历 ID。 aResource ：包含事件数据的对象。	创建一个新事件。
DeleteEvent	aCalendarId ：日历的 id。 aid ：事件的标识符。	删除现有事件。
UpdateEvent	aCalendarId ：日历的 id。 aResource ：包含事件数据的对象。	更新现有事件。
LoadEvents	aCalendarId ：日历的 id。	加载日历的所有事件。
LoadEventsChanged	aCalendarId ：日历的 ID。 aSyncToken ：用于更新日历的最后令牌。	从设置的 Token 加载日历的所有事件。

您可以使用 **Calendars** 属性访问事件数据。从列表中选择任何现有日历，并访问其 **Events** 属性。

Google Calendar API 调用

方法	描述
ACL_Delete	删除访问控制规则。
ACL_Get	返回访问控制规则。
ACL_Insert	创建访问控制规则。
ACL_List	返回日历的访问控制列表中的规则。
ACL_Patch	更新访问控制规则。此方法支持 patch 语义。
ACL_Update	更新访问控制规则。
ACL_Watch	监视 ACL 资源的变更。

方法	描述
CalendarList_Delete	从用户的日历列表中删除一个日历。
CalendarList_Get	从用户日历列表中返回一个日历。
CalendarList_Insert	将现有日历插入用户的日历列表。
CalendarList_List	返回用户日历列表中的日历。
CalendarList_Patch	更新用户日历列表中的现有日历。此方法支持 patch 语义。
CalendarList_Update	更新用户日历列表中的现有日历。
CalendarList_Watch	监视 CalendarList 资源的更改。

方法	描述
Calendar_Clear	清除主日历。此操作删除与账户主日历关联的所有事件。
Calendar_Delete	删除辅助日历。使用 calendars.clear 清除主日历上的所有事件。
Calendar_Get	返回日历的元数据。
Calendar_Insert	创建辅助日历。
Calendar_Patch	更新日历的元数据。此方法支持 patch 语义。
Calendar_Update	更新日历的元数据。

方法	描述
Channel_Stop	停止通过此频道监视资源。

方法	描述
Color_Get	返回日历和事件的颜色定义。

方法	描述
Event_Delete	删除一个事件。
Event_Get	返回一个事件。
Event_Import	导入事件。此操作用于将现有事件的私有副本添加到日历中。
Event_Insert	创建一个事件。
Event_Instances	返回指定重复事件的实例。

Event_List	返回指定日历上的事件。
Event_Move	将事件移到另一个日历，即更改事件的组织者。
Event_Patch	更新一个事件。此方法支持补丁语义。您指定的字段值将替换现有值。请求中未指定的字段保持不变。如果指定了数组字段，则会覆盖现有数组，从而丢弃之前的所有数组元素。
Event_QuickAdd	基于简单文本字符串创建一个事件。
Event_Update	更新事件。
Event_Watch	监视 Events 资源的变更。

方法	描述
Freebusy_Query	返回一组日历的空闲/忙碌信息。

方法	描述
Settings_Get	返回单个用户设置。
Settings_List	返回已认证用户的所有用户设置。
Settings_Watch	监视 Settings 资源的变更。

切换账户

如果要从一个账户切换到另一个账户，请先调用 **Clear** 方法清除当前日历会话，然后再配置新会话。

Google 日历 | 加载日历

从您的账户获取所有日历的过程非常简单。只需按照以下步骤操作：

1. 调用方法 **LoadCalendars**。
2. 如果该方法返回 **True**，则您可以访问 **Calendars** 属性并遍历列表以访问所有日历。

```
TsgcHTTPGoogleCloud_Calendar_Client oGoogleCalendar = new TsgcHTTPGoogleCloud_Calendar_Client(NULL);
// ... configure OAuth2 options
oGoogleCalendar->GoogleCloudOptions->OAuth2->ClientId = "google ClientId";
oGoogleCalendar->GoogleCloudOptions->OAuth2->ClientSecret = "google ClientSecret";
// ... request calendars
if (oGoogleCalendar->LoadCalendars)
{
    // ... get calendars data
    for (int i = 0; i < oGoogleCalendar->Calendars->Count; i++)
    {
        vCalendarTitle = oGoogleCalendar->Calendars->Calendar[i]->Summary;
    }
}
else
{
    throw Exception("Error Calendar Sync");
}
```

Google Calendar | Sync Events

从日历获取所有事件的过程非常简单。只需按照以下步骤操作：

1. 调用方法 **LoadEvents** 并将 **CalendarId** 作为参数传入。
2. 如果方法返回 True，则您可以访问 **Calendars.Events** 属性，并遍历列表以访问日历的所有事件。

```
TsgcHTTPGoogleCloud_Calendar_Client oGoogleCalendar = new TsgcHTTPGoogleCloud_Calendar_Client(NULL);
// ... configure OAuth2 options
oGoogleCalendar->GoogleCloudOptions->OAuth2->ClientId = "google ClientId";
oGoogleCalendar->GoogleCloudOptions->OAuth2->ClientSecret = "google ClientSecret";
// ... request calendars first;
oGoogleCalendar->LoadCalendars;
// ... request events from first calendar
oCalendar = TsgcGoogleCalendarItem(oGoogleCalendar->Calendars->Calendar[0]);
if (oGoogleCalendar->LoadEvents(oCalendar->ID))
{
    // ... get events data
    for (int i = 0; i < oCalendar->Events->Count; i++)
    {
        vEventTitle = oCalendar->Events[i]->Summary;
    }
}
else
{
    throw Exception("Error Event Sync");
}
```

Google Calendar | RefreshToken

Google Calendar API 使用 OAuth2 对 Google 服务器进行身份验证。sgcWebSockets 有一个处理整个身份验证流程的组件，但如果您的应用关闭后再次尝试连接，您有两个选择：

1. 使用您的 Google API 重新进行身份验证
2. 使用刷新令牌（若仍有效），以避免重复执行认证流程。

使用 RefreshToken

首次进行身份验证时，若 **RefreshToken** 存在，使用 OnAuthToken 事件保存它。例如，可将其保存在 INI 文件中：

```
void OnGoogleCalendarAuthToken(TObject *Sender, string TokenType, string Token, string Data)
{
    TsgcJSON *oJSON = new TsgcJSON();
    try
    {
        oJSON->Read(Data);
        if (oJSON->Node["refresh_token"] != NULL)
        {
            TINIFile *oINI = new TINIFile(ChangeFileExt(Application->ExeName, ".ini"));
            try
            {
                oINI->WriteString("OAUTH2", "Token", oJSON->Node["refresh_token"]->Value);
            }
            __finally
            {
                oINI->Free();
            }
        }
    }
    __finally
    {
        oJSON->Free();
    }
}
```

然后，当您再次启动应用程序时，如果存在 RefreshToken，请调用 RefreshToken 方法并将令牌作为参数传递（必须先设置 Google Calendar API 密钥）。如果成功，您将无需重新验证即可登录 Google 服务器。

```
GoogleCalendar->RefreshToken("your refresh token here");
```


Google Calendar | 服务账号

Google Calendar 客户端可以作为无需用户交互的服务运行，因此在需要运行 Windows 服务、守护进程等情况时非常有用。

Google Cloud 要求创建一个**服务账户**（而非 OAuth2 凭据）来运行此类项目，且 Google Calendar API 要求服务账户使用**全域委派**以获取访问日历所需的凭据。

您可以阅读有关如何创建 [Google 服务账号](#) 的更多信息。

一旦 Google Cloud 账号配置了服务账号并使用域范围委派链接到工作区电子邮件账号，您可以按照以下步骤配置 Google Calendar 客户端与其配合使用：

- 将属性 **GoogleCloudOptions.Authentication** 设置为值 **gcaJWT**。
- 使用方法 **LoadSettingsFromFile** 导入在您的 Google Cloud 账户中生成的 JSON 文件。该文件包含用于加密 JWT 的私钥以及客户端所需的其他属性。
- 导入 JSON 文件后，以下属性将自动填充：
 - **ClientEmail**：服务账户名称
 - **PrivateKeyId**：私钥文件的 ID
 - **PrivateKey**：私钥文件
- 最后，在属性 **GoogleCloudOptions.JWT.Subject** 中设置与服务账号关联的工作区电子邮件账户。

配置客户端后，无需用户交互即可开始向 Google Calendar API 发送请求。

Google Cloud FCM

Firebase Cloud Messaging (FCM) 是一种跨平台消息传递解决方案，可让您以零成本可靠地发送消息。

使用 FCM，您可以通知客户端应用程序有新邮件或其他数据可供同步。您可以发送通知消息以促进用户重新参与和留存。对于即时通讯等用例，消息可以向客户端应用程序传输最多 4096 字节的有效载荷。

该组件支持 HTTP v1 API。

授权

Google FCM 组件客户端可使用以下方法登录 Google 服务器：

- **gcaOAuth2**：OAuth2 协议
- **gcaJWT**：JWT 令牌。

OAuth2

登录通过 Web 浏览器完成，用户使用自己的账户登录并授权 FCM 请求。

- **GoogleCloudOptions.OAuth2.ClientId**：是 Google 为通过 OAuth2 协议进行身份验证所提供的 ClientID。
- **GoogleCloudOptions.OAuth2.ClientSecret**：Google 提供的客户端密钥字符串，用于通过 OAuth2 协议进行身份验证。
- **GoogleCloudOptions.OAuth2.Scope**：是 OAuth2 的范围，通常无需修改默认值，除非需要获取超出默认范围的访问权限。
- **GoogleCloudOptions.OAuth2.LocalIP**：OAuth2 协议需要一个本地服务器来监听来自身份验证服务器的响应。这是 IP 或 DNS 名称。默认情况下为 127.0.0.1。
- **GoogleCloudOptions.OAuth2.LocalPort**：本地服务器监听端口。
- **GoogleCloudOptions.OAuth2.RedirectURL**：如果您需要设置与 LocalPort + LocalIP 不同的重定向 URL，可以在此属性中设置（示例：<http://127.0.0.1:8080/oauth2>）。

服务账号

登录通过使用 Google 提供的私钥对请求进行签名来完成。此方法推荐用于无需用户交互的自动化服务或应用程序。

- **GoogleCloudOptions.JWT.ClientEmail**：创建新服务账户时提供的客户端邮箱名称。即 JSON 配置文件中的 "client_email" 节点。
- **GoogleCloudOptions.JWT.Subject**：创建新服务账号时提供的客户端电子邮件名称。对应 JSON 配置文件中的 "client_email" 节点。
- **GoogleCloudOptions.JWT.PrivateKeyId**：Google 提供的私钥 ID。JSON 配置文件中的 "private_key_id" 节点。
- **GoogleCloudOptions.JWT.PrivateKey**：Google 提供的私钥证书。JSON 配置文件中的 "private_key" 节点。

TLS 选项

Firebase Cloud Messaging 连接可通过 **TLSOptions** 属性进行调整。

- **TLSOptions.IOHandler**：选择将使用的 TLS 栈（OpenSSL、SChannel 等）。
- **TLSOptions.Version**：请求项目所需的特定 TLS 协议版本。
- **TLSOptions.VerifyCertificate**：连接到 Google 时强制执行服务器证书验证。
- **TLSOptions.OpenSSL_Options.LibPath**：定义 OpenSSL 运行时库的部署位置。
- **TLSOptions.SChannel_Options.UseLegacyCredentials**：当 Windows SChannel 需要旧版凭据协商时，启用此选项。

配置示例：

```
TsgcHTTPGoogleCloud_FCM_Client *oFCM = new TsgcHTTPGoogleCloud_FCM_Client(NULL);
oFCM->TLSOptions->IOHandler = ioHandlerOpenSSL;
oFCM->TLSOptions->Version = tls1_3;
oFCM->TLSOptions->VerifyCertificate = true;
oFCM->TLSOptions->OpenSSL_Options->LibPath = opensslDefaultFolder;
```

创建新服务账户后，您可以下载一个包含所有配置的 JSON 文件。该文件可由 FCM 组件处理，只需调用方法 **LoadSettingsFromFile** 并将 JSON 文件名作为参数传入即可。

最常见的用途

- 配置
 - [Google OAuth2 密钥](#)
 - [服务账号](#)

Google FCM 客户端

OAuth2

要使用 Google FCM API，sgcWebSockets FCM 组件使用 OAuth2 作为默认认证方式，因此首先需要设置您 Google 账户的 **ClientId** 和 **ClientSecret**。

```
oFCM = new TsgcHTTPGoogleCloud_FCM_Client();
oFCM->GoogleCloudOptions->Authorization = gcaOAuth2;
oFCM->GoogleCloudOptions->OAuth2->ClientId = "... your google client id...";
oFCM->GoogleCloudOptions->OAuth2->ClientSecret = "... your google client secret...";
```

服务账号

服务账号需要构建 JWT 并将其作为授权令牌传递。

```
TsgcHTTPGoogleCloud_FCM_Client *oFCM = new TsgcHTTPGoogleCloud_FCM_Client(this);
oFCM->GoogleCloudOptions->Authorization = gcaJWT;
oFCM->GoogleCloudOptions->JWT->ClientEmail = "...google email...";
oFCM->GoogleCloudOptions->JWT->Subject = "...google email...";
oFCM->GoogleCloudOptions->JWT->PrivateKeyId = "...private key id...";
oFCM->GoogleCloudOptions->JWT->PrivateKey->Lines->Text = "...private key certificate...";
```

这是从 Google 获取授权令牌密钥所必需的，该密钥将用于所有 REST API 调用。

所有方法都返回响应，该响应可能成功或返回错误。

发送消息

使用 **SendMessage** 方法发送通知消息。该函数有 2 个参数：

1. Project：是您的 Google Cloud 账户的项目 id。
2. Payload：JSON 文本消息。示例：

```
{
  "message": {
    "topic": "news",
    "notification": {
      "title": "Breaking News",
      "body": "New news story available."
    },
    "data": {
      "story_id": "story_12345"
    }
  }
}
```


TsgcWebView2

TsgcWebView2 是一个可视化 VCL 组件，封装了 Microsoft Edge WebView2（Chromium）。将其放置在窗体上，即可在 Delphi 应用程序中嵌入现代化的全功能 Web 浏览器。支持 Delphi 7 至 Delphi 13。

系统要求

- Microsoft Edge WebView2 Runtime（随 Windows 10/11 附带或可从 Microsoft 下载）
- WebView2Loader.dll（已包含，请放置于可执行文件旁边）
- 仅限 Windows

快速入门

将 TsgcWebView2 拖放到窗体上并导航到 URL：

```
sgcWebView21 := TsgcWebView2.Create(Self);
sgcWebView21.Parent := Self;
sgcWebView21.Align := alClient;
sgcWebView21.DefaultURL := 'https://www.example.com';
```

功能特性

- 导航
 - [Navigate](#)、[NavigateToString](#)、[GoBack](#)、[GoForward](#)、[Reload](#)、[Stop](#)、[POST 导航](#)
- JavaScript
 - [ExecuteScript](#)（异步）、[ExecuteScriptSync](#)、[AddInitScript](#)
- Cookie 管理
 - [GetCookies](#)、[AddOrUpdateCookie](#)、[DeleteCookie](#)、[DeleteAllCookies](#)
- 下载控制
 - [OnDownloadStarting](#)、[OnDownloadProgress](#)、[OnDownloadCompleted](#)
- 设置与配置
 - [ScriptEnabled](#)、[DevToolsEnabled](#)、[ContextMenuEnabled](#) 等
- 高级功能
 - [打印](#)、[音频/静音](#)、[证书](#)、[Favicon](#)、[虚拟主机](#)、[截图](#)

属性

- **DefaultURL**：string — 初始化后导航到的 URL。
- **AutoInitialize**：Boolean — 当值为 True 时，窗口句柄分配后将自动创建 WebView2。默认值：True。
- **ZoomFactor**：Double — 缩放级别（1.0 = 100%）。
- **UserDataFolder**：字符串，用于存储 Cookie、缓存和权限的路径。
- **BrowserExecutableFolder**：string，指向固定版本 WebView2 Runtime 发行版的路径。
- **Settings**：TsgcWebView2Settings — 设计时浏览器设置（ScriptEnabled、DevToolsEnabled 等）。
- **URL**：字符串（只读）— 当前页面 URL。
- **DocumentTitle**：字符串（只读）——当前页面标题。
- **CanGoBack**：Boolean（只读），表示是否可以后退导航。
- **CanGoForward**：Boolean（只读）——是否可以向前导航。
- **Initialized**：Boolean（只读）— WebView2 是否已就绪。
- **IsMuted**：Boolean — 静音/取消静音音频。
- **IsDocumentPlayingAudio**：Boolean（只读）— 页面是否正在播放音频。
- **FaviconURI**：字符串（只读）— 当前页面图标的 URI。
- **StatusBarText**：string（只读）— 当前状态栏文本。
- **CookieManager**：TsgcWebView2CookieManager（只读）— Cookie 管理对象。
- **WebView**：ICoreWebView2（只读）— 直接 COM 接口访问。
- **Controller**：ICoreWebView2Controller（只读）— 直接 COM 接口访问。
- **Environment**：ICoreWebView2Environment（只读）— 直接 COM 接口访问。

方法

- **Navigate**(aURL: string) — 导航到 URL。
- **NavigateToString**(aHTML: string) — 直接加载 HTML 内容。
- **GoBack / GoForward** — 导航历史记录。
- **Reload / Stop** — 重新加载或停止加载。
- **ExecuteScript**(aScript: string) — 异步执行 JavaScript (结果通过 OnScriptExecuted 返回)。
- **ExecuteScriptSync**(aScript: string): string — 同步执行 JavaScript, 返回 JSON 结果。
- **AddInitScript**(aScript: string) — 添加在每次页面加载时运行的 JavaScript。
- **RemoveInitScript**(aScriptId: string) — 删除之前添加的初始化脚本。
- **NavigateWithPostData**(aURI, aMethod, aPostData, aHeaders: string), 使用自定义 HTTP 方法、请求体和标头进行导航。
- **PrintToPdf**(aFilePath: string) — 将页面保存为 PDF。
- **ShowPrintUI** — 显示原生打印对话框。
- **CapturePreviewToFile**(aFilePath: string; aFormat: Integer) — 截图 (0=PNG, 1=JPEG)。
- **OpenDevToolsWindow** — 打开 Edge 开发者工具。
- **OpenTaskManagerWindow** — 打开 Edge 任务管理器。
- **SetVirtualHostNameToFolderMapping**(aHostName, aFolderPath: string; aAccessKind: Integer) — 将主机名映射到本地文件夹。
- **ClearVirtualHostNameToFolderMapping**(aHostName: string) — 移除映射。
- **PostWebMessageAsString**(aMessage: string) — 向 Web 内容发送消息。
- **PostWebMessageAsJson**(aJson: string) — 向 Web 内容发送 JSON。
- **GetProfileName**: string — 获取浏览器配置文件名称。
- **ClearBrowsingData**(aKinds: Cardinal) — 清除特定类型的浏览数据。
- **ClearAllBrowsingData** — 清除所有浏览数据。
- **PostSharedBufferToScript**(aSharedBuffer: IUnknown; aAccess: Integer; aAdditionalDataAsJson: string) — 与 Web 内容共享内存缓冲区。
- **InitializeWebView / FinalizeWebView** — 手动初始化控制。

事件

- **OnInitialized** — 当 WebView2 就绪时触发。
- **OnNavigationStarting**(aURI; alsUserInitiated, alsRedirected: Boolean; var aCancel: Boolean) — 导航开始前触发。
- **OnNavigationCompleted**(alsSuccess: Boolean; aWebErrorStatus: Integer) — 导航完成后。
- **OnSourceChanged**(alsNewDocument: Boolean) — URL 已更改。
- **OnDocumentTitleChanged**(aTitle: string) — 页面标题已更改。
- **OnWebMessageReceived**(aSource, aWebMessageAsJson, aWebMessageAsString: string) — 来自 Web 内容的消息。
- **OnNewWindowRequested**(aURI: string; alsUserInitiated: Boolean; var aHandled: Boolean) — 请求弹出窗口/新标签页。
- **OnScriptExecuted**(aErrorCode: HRESULT; aResultAsJson: string) — 异步 JS 结果。
- **OnContentLoading**(alsErrorPage: Boolean) — 内容开始加载。
- **OnHistoryChanged** — 导航历史记录已更改。
- **OnProcessFailed**(aKind: Integer) — 浏览器进程崩溃。
- **OnPermissionRequested**(aURI: string; aKind: Integer; alsUserInitiated: Boolean; var aState: Integer) — 权限请求。
- **OnWindowCloseRequested** — 调用了 window.close()。
- **OnDownloadStarting**(aURI, aResultFilePath: string; var aCancel, aHandled: Boolean; var aFilePath: string) — 下载已开始。
- **OnDownloadProgress**(aBytesReceived, aTotalBytes: Int64) — 下载进度。
- **OnDownloadCompleted**(aFilePath: string; aState: Integer) — 下载已完成。
- **OnContextMenuRequested**(aMenuItems: string; aContextKind: Integer; aLocation: TPoint; var aHandled: Boolean) — 右键菜单。
- **OnFaviconChanged**(aFaviconURI: string) — 页面图标已更改。
- **OnClientCertificateRequested**(aHost: string; aPort: Integer; var aHandled: Boolean) — 需要客户端证书。
- **OnServerCertificateError**(aRequestURI: string; aErrorStatus: Integer; var aAction: Integer) — 证书错误。
- **OnStatusBarTextChanged**(aText: string) — 状态栏文本已更改。
- **OnDOMContentLoaded** — DOM 就绪。
- **OnBasicAuthRequested**(aURI: string; var aUserName, aPassword: string; var aHandled: Boolean) — HTTP 基本认证。
- **OnCapturePreviewCompleted**(aErrorCode: HRESULT) — 截图完成。

TsgcWebView2 | 导航

TsgcWebView2 提供了多种方法，用于导航到 URL、加载 HTML 内容以及控制浏览器历史记录。

基本导航

使用 **Navigate** 方法加载 URL。组件在请求前触发 **OnNavigationStarting**，完成时触发 **OnNavigationCompleted**。

```
// Navigate to a URL
sgcWebView21.Navigate('https://www.example.com');
```

您还可以在设计时或初始化之前设置 **DefaultURL** 属性。WebView2 准备好后，组件将自动导航到此 URL。

```
// Set the default URL before initialization
sgcWebView21.DefaultURL := 'https://www.example.com';
```

HTML 内容

使用 **NavigateToString** 可直接加载 HTML 内容，无需服务器或文件。

```
// Load HTML content directly
sgcWebView21.NavigateToString(
  '<html><body><h1>Hello from Delphi</h1>' +
  '<p>This content was loaded with NavigateToString.</p>' +
  '</body></html>');
```

历史导航

使用 **GoBack** 和 **GoForward** 浏览历史记录。在调用这些方法之前，请先检查 **CanGoBack** 和 **CanGoForward**。

```
// Navigate back
if sgcWebView21.CanGoBack then
  sgcWebView21.GoBack;

// Navigate forward
if sgcWebView21.CanGoForward then
  sgcWebView21.GoForward;
```

使用 **Reload** 刷新当前页面，使用 **Stop** 取消待处理的导航。

```
// Reload the current page
sgcWebView21.Reload;

// Stop loading
sgcWebView21.Stop;
```

POST 导航

使用 **NavigateWithPostData** 发送带有自定义方法、请求体和标头的 HTTP 请求。这对于直接在浏览器中提交表单数据或调用 REST API 非常有用。

```
// 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');
```

导航事件

使用 **OnNavigationStarting** 在导航开始前检查或取消导航。将 **aCancel** 设为 **True** 以阻止该请求。

```
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;
```

使用 **OnNavigationCompleted** 检查导航是否成功或失败。

```
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;
```

新窗口处理

当页面打开弹出窗口或带有 **target="_blank"** 的链接时，将触发 **OnNewWindowRequested** 事件。将 **aHandled** 设置为 **True** 可阻止默认弹出并在同一窗口中导航。

```
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 提供了在浏览器上下文中执行 JavaScript、检索结果以及在您的 Delphi 应用程序与 Web 内容之间建立双向通信的方法。

异步执行

使用 **ExecuteScript** 异步运行 JavaScript。结果将以 JSON 字符串的形式在 **OnScriptExecuted** 事件中返回。

```
// 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;
```

您可以执行任何有效的 JavaScript 表达式。返回值始终序列化为 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"');
```

同步执行

使用 **ExecuteScriptSync** 可进行阻塞执行并直接返回结果。当您需要立即获取值时，这种方式更简单，但会阻塞调用线程直到脚本执行完毕。

```
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;
```

初始化脚本

使用 **AddInitScript** 注册在每次页面加载时自动运行的 JavaScript，在页面上的任何其他脚本之前执行。这对于注入 polyfill、覆盖浏览器 API 或设置消息处理程序非常有用。

```
// 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);' +
'};');
```

使用 **RemoveInitScript** 通过 ID 注销之前添加的初始化脚本。

Web 消息传递

Web 消息提供 Delphi 应用程序与浏览器中运行的 JavaScript 之间的双向通信。使用 **PostWebMessageAsString** 或 **PostWebMessageAsJson** 从 Delphi 向 JavaScript 发送数据，并处理 **OnWebMessageReceived** 以接收来自 JavaScript 的数据。

从 Delphi 向 JavaScript 发送消息：

```
// Send a plain string message
sgcWebView21.PostWebMessageAsString('Hello from Delphi!');

// Send a JSON message
sgcWebView21.PostWebMessageAsJson('{"action":"refresh","id":42}');
```

在 JavaScript 中接收消息：

```
// 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);' +
'});');
```

在 Delphi 中从 JavaScript 接收消息：

```
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;
```

在 JavaScript 中，使用以下方式向 Delphi 发送消息：

```
// 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 管理

TsgcWebView2 公开了一个 **CookieManager** 属性，用于在浏览器上下文中读取、创建、更新和删除 Cookie。

获取 Cookie

使用 **CookieManager.GetCookies** 检索指定 URI 的 Cookie。该方法返回您可以遍历的 Cookie 对象列表。

```
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;
```

添加 Cookie

使用 **CookieManager.AddOrUpdateCookie** 创建新 Cookie 或更新现有 Cookie。至少需要指定名称、值、域和路径。

```
// 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
```

删除 Cookie

使用 **DeleteCookie** 按名称和 URI 删除特定 Cookie。使用 **DeleteAllCookies** 清除浏览器中的所有 Cookie。

```
// 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 | 下载控制

TsgcWebView2 提供事件以拦截、监控和控制浏览器发起的文件下载。

下载事件

OnDownloadStarting 事件在下载开始时触发。您可以取消下载、更改目标文件路径，或按默认行为继续。

```
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.');
```

下载进度

OnDownloadProgress 事件在下载期间定期触发，报告已接收的字节数和预期的总字节数。

```
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;
```

下载完成

OnDownloadCompleted 事件在下载完成时触发。检查 **aState** 以确定下载是否成功、已取消或已失败。

```
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.');
```

自定义下载路径

在 **OnDownloadStarting** 中设置 **aFilePath** 参数，可将下载重定向到自定义位置。将 **aHandled** 设为 **True** 可抑制默认的保存对话框。

```
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 | 设置

TsgcWebView2 公开了一个类型为 TsgcWebView2Settings 的 **Settings** 属性，用于控制浏览器行为。这些设置可以在设计时配置，也可以在初始化后的运行时更改。

TsgcWebView2Settings 属性

- **ScriptEnabled** : Boolean — 启用或禁用 JavaScript 执行。默认：True。
- **WebMessageEnabled** : Boolean — 启用或禁用 Web 消息传递 (postMessage)。默认值：True。
- **DefaultScriptDialogsEnabled** : 布尔值，启用或禁用默认 JavaScript 对话框 (alert、confirm、prompt)。默认值：True。
- **StatusBarEnabled** : Boolean — 显示或隐藏浏览器状态栏。默认值：True。
- **DevToolsEnabled** : Boolean，允许或阻止访问 Edge 开发者工具。默认值：True。
- **ContextMenuEnabled** : 布尔值——启用或禁用浏览器右键上下文菜单。默认：True。
- **ZoomControlEnabled** : Boolean，允许或阻止用户发起的缩放操作 (Ctrl+滚轮、Ctrl+加/减)。默认值：True。
- **BuiltInErrorPageEnabled** : Boolean，显示或隐藏导航失败的内置错误页面。默认：True。

设计时配置

在对象检查器中设置 Settings 属性。WebView2 环境初始化后，这些属性将自动应用。

```
// 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;
```

运行时配置

您可以在组件初始化后在运行时更改设置。更改会立即生效。

```
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

UserDataFolder 属性指定浏览器存储 Cookie、缓存、权限及其他配置文件数据的目录。每个唯一的文件夹创建一个独立的浏览器配置文件。

```
// Use a custom data folder for isolated profiles
sgcWebView21.UserDataFolder := 'C:\AppData\MyApp\Profile1';
```

在初始化之前设置此属性（在组件创建 WebView2 环境之前）。如果未设置，将在应用程序的临时目录中使用默认文件夹。

BrowserExecutableFolder

BrowserExecutableFolder 属性指向固定版本的 WebView2 Runtime 发行版。使用此属性可随应用程序一起发布特定浏览器版本，而无需依赖系统安装的运行时。

```
// Use a fixed-version runtime bundled with the application
sgcWebView21.BrowserExecutableFolder :=
  ExtractFilePath(ParamStr(0)) + 'WebView2Runtime';
```

在初始化之前设置此属性。如果留空，组件将使用系统安装的 Evergreen WebView2 Runtime。

TsgcWebView2 | 高级功能

TsgcWebView2 提供对高级浏览器功能的访问，包括打印、截图、音频控制、证书处理等。

打印支持

使用 **PrintToPdf** 将当前页面保存为 PDF 文件，或使用 **ShowPrintUI** 显示原生打印对话框。

```
// Save the current page as PDF
sgcWebView21.PrintToPdf('C:\Output\page.pdf');
```

```
// Show the native print dialog
sgcWebView21.ShowPrintUI;
```

截图捕获

使用 **CapturePreviewToFile** 对当前页面进行截图。format 参数指定图像类型：0 为 PNG，1 为 JPEG。截图完成时触发 **OnCapturePreviewCompleted** 事件。

```
// 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;
```

音频/静音控制

使用 **IsMuted** 属性静音或取消静音音频播放。检查 **IsDocumentPlayingAudio** 以检测页面当前是否正在播放音频。

```
// Toggle mute
sgcWebView21.IsMuted := not sgcWebView21.IsMuted;

// Check if audio is playing
if sgcWebView21.IsDocumentPlayingAudio then
  LabelStatus.Caption := 'Audio is playing';
```

证书处理

使用 **OnClientCertificateRequested** 响应服务器需要客户端证书的情况。使用 **OnServerCertificateError** 处理 TLS 证书错误（例如开发环境中的自签名证书）。

```
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;
```

虚拟主机映射

使用 **SetVirtualHostNameToFolderMapping** 将主机名映射到本地文件夹。这样，Web 内容就可以使用虚拟 URL 引用本地文件，而非 file:// 路径。

```
// 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');
```

配置文件管理

使用 **GetProfileName** 获取当前浏览器配置文件名称。使用 **ClearBrowsingData** 或 **ClearAllBrowsingData** 删除缓存数据、Cookie 及其他浏览记录。

```
// Get the profile name
ShowMessage('Profile: ' + sgcWebView21.GetProfileName);

// Clear all browsing data
sgcWebView21.ClearAllBrowsingData;

// Clear specific browsing data types
sgcWebView21.ClearBrowsingData($00001); // cache only
```

基本认证

当服务器请求 HTTP Basic 身份验证时，触发 **OnBasicAuthRequested** 事件。提供凭据并将 **aHandled** 设置为 True。

```
procedure TFormMain.sgcWebView21BasicAuthRequested(Sender: TObject;
  const aURI: string; var aUserName, aPassword: string;
  var aHandled: Boolean);
begin
  aUserName := 'admin';
  aPassword := 'secret';
  aHandled := True;
end;
```

上下文菜单

当用户在浏览器中单击鼠标右键时，触发 **OnContextMenuRequested** 事件。将 **aHandled** 设置为 True 可禁止默认菜单并显示您自己的菜单。

```
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

FaviconURI 属性返回当前页面 favicon 的 URI。**OnFaviconChanged** 事件在 favicon 改变时触发。

```
procedure TFormMain.sgcWebView21FaviconChanged(Sender: TObject;
  const aFaviconURI: string);
begin
  LabelFavicon.Caption := 'Favicon: ' + aFaviconURI;
end;
```

状态栏

StatusBarText 属性返回当前状态栏文本（通常是悬停链接的 URL）。**OnStatusBarTextChanged** 事件在文本更改时触发。

```
procedure TFormMain.sgcWebView21StatusBarTextChanged(Sender: TObject;
  const aText: string);
begin
  StatusBar1.SimpleText := aText;
end;
```

任务管理器

使用 **OpenTaskManagerWindow** 打开 Edge 浏览器任务管理器，显示每个浏览器进程的内存和 CPU 使用情况。

```
sgcWebView21.OpenTaskManagerWindow;
```

共享缓冲区

使用 **PostSharedBufferToScript** 在您的 Delphi 应用程序与 Web 内容之间共享内存缓冲区，以实现高性能数据传输。

```
// 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}');
```

直接 COM 访问

对于组件 API 未涵盖的高级场景，可使用 **WebView**、**Controller** 和 **Environment** 属性直接访问底层 WebView2 COM 接口。

```
var
  vWebView: ICoreWebView2;
begin
  vWebView := sgcWebView21.WebView;
  if Assigned(vWebView) then
  begin
    // Call any ICoreWebView2 method directly
  end;
end;
```

TsgcUDPClient

TsgcUDPClient 基于 Indy 库实现 UDP 客户端。

简介

TsgcUDPClient 基于 Indy 库实现 UDP 客户端。

UDP 是一种无连接协议，不保证发送的消息到达目的地，但与 TCP 协议相比，它的速度要快得多。

1. 在表单上放置一个 **TsgcUDPClient** 组件
2. 将 **Host** 和 **Port**（默认为 80）设置为连接到可用的 UDP 服务器。

```
oClient = new TsgcUDPClient();  
oClient->Host = "127.0.0.1";  
oClient->Port = 80;
```

3. 您可以通过 HTTP 代理服务器连接，需要定义代理属性：

Host：代理服务器主机名。

Port：代理服务器端口号。

Username：认证用的用户名，匿名访问留空。

密码：用于认证的密码，匿名登录时留空。

4. 如果需要，您可以处理事件
5. 调用 **WriteData** 方法向 UDP 服务器发送任意消息。

参考

- 属性 — 该组件提供 9 个属性。
- 方法 — 2 个公共方法。
- 事件 — 3 个可处理的事件。

TsgcUDPServer

TsgcUDPServer 基于 Indy 库实现 UDP 服务器。

简介

TsgcUDPServer 基于 Indy 库实现 UDP 服务器。

UDP 是一种无连接协议，不保证消息能到达目的地，但与 TCP 协议相比，速度要快得多。

1. 将 **TsgcUDPServer** 组件拖放到窗体上

2. 设置监听端口。

```
oClient = new TsgcUDPServer();  
oClient->Port = 80;
```

3. 要启动服务器，请将属性 **Active = true** 设置为 true。

参考

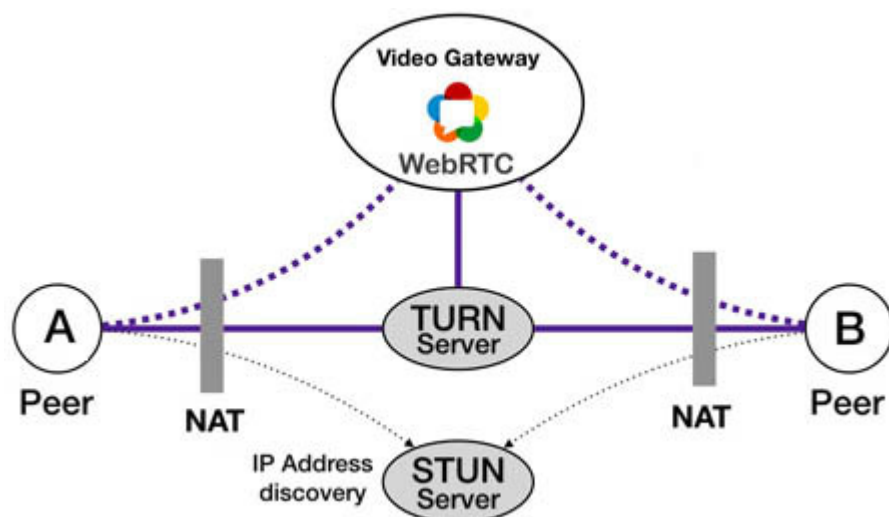
- 属性 — 组件上提供 11 个属性。
- 方法 — 7 个公共方法。
- 事件 — 6 个可处理的事件。

STUN

STUN（NAT 会话穿越实用工具）是一种 IETF 协议，用于 IP 网络中的实时音视频通信。STUN 采用服务器-客户端架构，STUN 服务器通常同时监听 UDP 和 TCP，默认端口为 3478。

STUN 协议的主要目的是使运行在 NAT 后面的设备能够发现其公共 IP 及其所在 NAT 的类型。

STUN 提供了一种在 NAT 后的对端之间进行通信的机制。对端向 STUN 服务器发送请求以获知公网 IP 地址和端口。客户端向服务器发送的绑定请求用于确定 NAT 分配的 IP 和端口绑定。STUN 客户端向 STUN 服务器发送绑定请求，服务器检查客户端使用的源 IP 和端口，并将此信息返回给客户端。



STUN 服务器基本上发送两种类型的响应：成功或错误。每个响应都有一个属性列表，其中包含有关绑定 IP 地址、错误代码、错误原因等的信息。

组件

- **TsgcSTUNClient**：实现 STUN 协议的客户端组件，允许您向 STUN 服务器发送绑定请求。
- **TsgcSTUNServer**：实现 STUN 协议的服务器端组件。

TsgcSTUNClient

TsgcSTUNClient 组件。

简介

TsgcSTUNClient 是实现 STUN 协议的客户端，允许您向 STUN 服务器发送绑定请求。

该组件允许使用 **UDP** 和 **TCP** 作为传输层。使用 UDP 作为传输层时，实现了**重传机制**，在短时间内未收到响应时重新发送请求。

基本用法

通常 STUN 服务器运行在 UDP 3478 端口上且不需要认证，因此要发送 STUN 绑定请求，请填写服务器属性以便客户端知道连接位置，并处理组件从服务器接收响应时的相关事件。

配置服务器

- Host：服务器的 IP 或 DNS 名称，例如：stun.sgcwebsockets.com
- Port：监听服务器端口，示例：3478

调用方法 **SendRequest**，向 STUN 服务器发送绑定请求。

处理事件

- 如果服务器返回成功响应，将调用事件 **OnSTUNResponseSuccess**，您可以通过读取 **aBinding** 对象访问绑定信息。
- 如果服务器返回错误，则会调用事件 **OnSTUNResponseError**，您可以通过读取 **aError** 对象来访问错误代码和原因。

```
TsgcSTUNClient oSTUN = new TsgcSTUNClient(this);
oSTUN->Host = "stun.sgcwebsockets.com";
oSTUN->Port = 3478;
oSTUN->SendRequest();

private void OnSTUNResponseSuccess(TObject *Sender, const TsgcSocketConnection *aSocket,
const TsgcSTUN_Message *aMessage, const TsgcSTUN_ResponseBinding *aBinding)
{
    DoLog("Remote IP: " + aBinding->RemoteIP + ". Remote Port: " + IntToStr(aBinding->RemotePort));
}

private void OnSTUNResponseError(TObject *Sender, const TsgcSocketConnection *aSocket,
const TsgcSTUN_Message *aMessage, const TsgcSTUN_ResponseError *aError)
{
    DoLog("Error: " + IntToStr(aError->Code) + " " + aError->Reason);
}
```

参考

- 属性 — 组件上提供 9 个属性。
- 方法 — 3 个公开方法。
- 事件——可处理 4 个事件。

指南

- 绑定
 - [UDP 重传](#)
 - [长期凭据](#)
- - [属性](#)

STUN 客户端 | UDP 重传

在 **UDP** 上运行 **STUN** 时，**STUN 消息** 可能会被网络丢弃。STUN 请求/响应事务的可靠性通过客户端应用程序本身重传请求消息来实现。

客户端应从 RTO（“重传超时”）间隔开始重传 STUN 请求消息，每次重传后加倍。RTO 是往返时间的估算值。

默认情况下，sgcWebSockets STUN 客户端已配置为 RTO 500 ms 和最大重试次数 7 次。

例如，假设 RTO 为 500 毫秒，请求将在 0 毫秒、500 毫秒、1500 毫秒、3500 毫秒、7500 毫秒、15500 毫秒和 31500 毫秒时发出。如果客户端在 39500 毫秒后仍未收到响应，则认为事务已超时。

```
TsgcSTUNClient oSTUN = new TsgcSTUNClient(this);
oSTUN->Host = "stun.sgcwebsockets.com";
oSTUN->Port = 3478;
oSTUN->RetransmissionOptions->Enabled = true;
oSTUN->RetransmissionOptions->RTO = 500;
oSTUN->RetransmissionOptions->MaxRetries = 7;
oSTUN->SendRequest();
```


STUN 客户端 | 长期凭据

长期凭据机制依赖于一种长期凭据，以用户名和密码的形式在客户端 和服务器之间共享。该凭据被认为是长期的，因为它假定为用户配置，并在用户不再是系统订阅者或凭据被更改之前一直有效。

您可以使用以下代码在 sgcWebSockets STUN 客户端中配置长期凭据。

```
TsgcSTUNClient oSTUN = new TsgcSTUNClient(this);
oSTUN->Host = "stun.sgcwebsockets.com";
oSTUN->Port = 3478;
oSTUN->STUNOptions->Authentication->Credentials = stauLongTermCredential;
oSTUN->STUNOptions->Authentication->Username = "user_name";
oSTUN->STUNOptions->Authentication->Password = "secret";
oSTUN->SendRequest();
```

如果服务器需要长期凭证而客户端发送的凭证有误，客户端将在 **OnSTUNResponseError** 事件中收到 401 Unauthorized 错误响应。

STUN 客户端 | 属性

每次服务器向客户端发送消息作为请求绑定的响应消息时，STUN 消息均包含一个属性列表，其中包含有关响应的信息。

您可以使用 `TsgcSTUN_Message` 类及其 `Attributes` 属性来访问这些属性，该属性包含一个包含有用信息的 `TsgcSTUN_Attribute` 对象列表。

```
private void OnSTUNResponseSuccess(TObject * Sender, const TsgcSocketConnection *aSocket,
    const TsgcSTUN_Message *aMessage, const TsgcSTUN_ResponseBinding *aBinding)
{
    DoLog("#binding: " + aBinding->RemoteIP + ":" + IntToStr(aBinding->RemotePort));

    for (int i = 0; i < aMessage->Attributes->Count; i++)
    {
        switch (TsgcSTUN_Attribute(aMessage->Attributes->Items[i])->AttributeType)
        {
            stmaFingerprint:
                DoLog("#fingerprint: " + IntToStr(dynamic_cast<TsgcSTUN_Attribute_FINGERPRINT*>
                    (aMessage->Attributes->Items[i])->Fingerprint));
            stmaSoftware:
                DoLog("#software: " + dynamic_cast<TsgcSTUN_Attribute_SOFTWARE*>
                    (aMessage->Attributes->Items[i])->Software);
            stmaResponse-Origin:
                DoLog("#response_origin: " + dynamic_cast<TsgcSTUN_Attribute_RESPONSE_ORIGIN*>
                    (aMessage->Attributes->Items[i])->Address + ":" +
                    IntToStr(dynamic_cast<TsgcSTUN_Attribute_RESPONSE_ORIGIN*>(aMessage->Attributes->Items
                        [i])->Port));
        }
    }
}
```

TsgcSTUNServer

STUN 服务器可以配置或不配置身份验证，可以验证指纹属性、发送备用服务器等。

简介

TsgcSTUNServer 是实现 STUN 协议的服务器，允许您处理来自 STUN 客户端的绑定请求。

STUN 服务器可以配置或不配置身份验证，可以验证指纹属性、发送备用服务器等。

基本用法

STUN 服务器通常运行在 UDP 端口 3478，且不需要身份验证，因此配置 STUN 服务器时，请设置侦听端口（默认 3478）并启动服务器。

配置服务器

- Port：监听服务器端口，示例：3478

将属性 **Active = True** 设置为启动 STUN 服务器。

```
TsgcSTUNServer oSTUN = new TsgcSTUNServer();  
oSTUN->Port = 3478;  
oSTUN->Active = true;
```

参考

- 属性 — 组件上可用的 8 个属性。
- 方法 — 2 个公共方法。
- 事件 — 4 个可处理的事件。

指南

- 配置
 - [长期凭据](#)
 - [备用服务器](#)

STUN 服务器 | 长期凭据

通常 STUN 服务器不配置身份验证，因此任何 STUN 客户端均可发送绑定请求，并期望在无需身份验证的情况下收到服务器响应。

sgcWebSockets STUN 服务器支持长期凭据，您可以配置 TsgcSTUNServer 使其只允许携带长期凭据信息的绑定请求。

要配置此功能，请访问 STUNOptions.Authorization 属性并启用它。
然后访问 LongTermCredentials 属性并启用它。默认情况下，此授权类型已配置了 Realm 字符串，并将默认 StaleNonce 值设为 10 分钟 (= 600 秒)。

```
TsgcSTUNServer oSTUN = new TsgcSTUNServer();
oSTUN->STUNOptions->Authentication->Enabled = true;
oSTUN->STUNOptions->Authentication->LongTermCredentials->Enabled = true;
oSTUN->STUNOptions->Authentication->LongTermCredentials->Realm = "sgcWebSockets";
oSTUN->STUNOptions->Authentication->LongTermCredentials->StaleNonce = 600;
oSTUN->Port = 3478;
oSTUN->Active = true;

private void OnSTUNRequestAuthorization(TObject *Sender, const TsgcSTUN_Message *aRequest,
const string aUsername, const string aRealm, ref string Password)
{
    if (aUsername == "my-user")
    {
        Password = "my-password";
    }
}
```

STUN 服务器 | 备用服务器

备用服务器表示一个备用传输地址，标识 STUN 客户端应尝试的不同 STUN 服务器。

STUN 服务器可配置为在响应绑定请求时发送备用服务器。要配置此行为，只需访问 `STUNOptions.BindingAttributes.AlternateServer` 属性并在此配置所需的值。

```
TsgcSTUNServer oSTUN = new TsgcSTUNServer();
oSTUN->Port = 3478;
oSTUN->STUNOptions->BindingAttributes->AlternateServer->Enabled = true;
oSTUN->STUNOptions->BindingAttributes->AlternateServer->IPAddress = "80.54.54.1";
oSTUN->STUNOptions->BindingAttributes->AlternateServer->Port = 3478;
oSTUN->Active = true;
```

当客户端收到 Alternate Server 响应属性时，将尝试向新服务器发送绑定请求。

TURN

穿越 NAT 的中继使用 (TURN) 协议使服务器能够在设备之间中继数据包。

如果主叫方和被叫方的公共 IP 地址均未被发现，TURN 提供一种回退技术，在端点之间中继通话。

连接 WebRTC 会话是一项需要多个 WebRTC 服务器协助完成的协调工作。WebRTC 中的 NAT 穿越服务器负责确保媒体正确连接。这些服务器是 STUN 和 TURN。

WebRTC 会话的连接方式

直接

如果两个设备都在本地网络上，则无需特殊操作即可建立连接。如果一个设备拥有另一个设备的本地 IP 地址，则它们可以直接相互通信。

直接使用公网 IP 地址

通过 [STUN](#) 协议获取的公网 IP 地址直接连接 WebRTC。

通过 TURN 服务器路由

当对等方位于 NAT 后且存在防火墙时，无法直接建立连接，因此需要 TURN 服务器在对等方之间路由数据。

组件

- [TsgcTURNClient](#)：实现 TURN 协议的客户端组件，允许您向 TURN 服务器执行分配、创建权限、发送指示等操作。
- [TsgcTURNServer](#)：实现 TURN 协议的服务器组件。

TsgcTURNClient

TsgcTURNClient 组件。

简介

TsgcTURNClient 是实现 TURN 协议的客户端，允许您向 TURN 服务器发送分配请求。该客户端继承自 STUN 客户端，因此 [STUN 客户端](#) 支持的所有方法均已在 TURN 客户端中支持。

基本用法

通常 TURN 服务器运行在 UDP 端口 3478 上，不需要身份验证，因此若要发送 TURN 请求，请填写服务器属性以告知客户端连接位置，并处理组件从服务器接收响应的相关事件。

配置服务器

- Host：服务器的 IP 或 DNS 名称，示例：turn.sgcwebsockets.com
- Port：监听服务器端口，示例：3478

调用 **Allocate** 方法，向 TURN 服务器发送请求以分配 IP 地址和端口。

处理事件

- 如果服务器返回成功响应，将调用 **OnTURNAllocateSuccess** 事件，您可以通过读取 **aAllocation** 对象来访问分配信息。
- 如果服务器返回错误，则会调用事件 **OnSTUNResponseError**，您可以通过读取 **aError** 对象来访问错误代码和原因。

```
TsgcTURNClient oTURN = new TsgcTURNClient(this);
oTURN->Host = "turn.sgcwebsockets.com";
oTURN->Port = 3478;
oTURN->Allocate();

private void OnTURNAllocate(TObject *Sender, const TsgcSocketConnection *aSocket,
    const TsgcSTUN_Message *aMessage, const TsgcTURN_ResponseAllocation *aAllocation)
{
    DoLog("Relayed IP: " + aAllocation->RelayedIP + ". Relayed Port: " + IntToStr(aAllocation->RelayedPort));
}

private void OnSTUNResponseError(TObject *Sender, const TsgcSTUN_Message *aMessage,
    const TsgcSTUN_ResponseError *aError)
{
    DoLog("Error: " + IntToStr(aError->Code) + " " + aError->Reason);
}
```

参考

- 属性 — 该组件提供 10 个属性。
- 方法 — 9 个公共方法。
- 事件——可处理 11 个事件。

指南

- 分配
 - [分配 IP 地址](#)
 - [创建权限](#)

- 指示
 - [发送指示](#)
- 频道
 - [TURN 客户端频道](#)

TURN 客户端 | 分配 IP 地址

TURN 协议允许使用中继 IP 地址在位于 NAT 后的对端之间交换数据。

要在 TURN 服务器上创建新的中继 IP 地址，客户端必须首先调用方法 **Allocate**，此方法向 TURN 服务器发送请求以创建新的中继 IP 地址，如果 TURN 服务器能够创建新的中继 IP 地址，客户端将收到成功响应。客户端可以在分配生命周期内定义的时间内与其他对等方通信。

```
TsgcTURNClient oTURN = new TsgcTURNClient(this);
oTURN->Host = "turn.sgcwebsockets.com";
oTURN->Port = 3478;
oTURN->Allocate();

private void OnTURNAllocate(TObject *Sender, const TsgcSocketConnection *aSocket,
const TsgcSTUN_Message *aMessage, const TsgcTURN_ResponseAllocation *aAllocation)
{
    DoLog("Relayed IP: " + aAllocation->RelayedIP + ". Relayed Port: " +
        IntToStr(aAllocation->RelayedPort));
}

private void OnSTUNResponseError(TObject *Sender, const TsgcSTUN_Message *aMessage,
const TsgcSTUN_ResponseError *aError)
{
    DoLog("Error: " + IntToStr(aError->Code) + " " + aError->Reason);
}
```

可使用 **Refresh** 方法更新生命周期以避免过期。Lifetime 是到期的秒数。若值为零，Allocation 将被删除。

```
oTURN->Refresh(600);
```

TURN 客户端 | 创建权限

在 TURN 服务器中创建新的分配后，如果没有权限，则该分配无法处理来自其他对端的任何入站数据包。因此，为允许其他对端使用中继 IP 地址进行通信，TURN 客户端必须首先为允许交换数据的 IP 地址创建权限。

要创建新权限，只需调用方法 **CreatePermission** 并将对等方的 IP 地址作为参数传入。如果对等方 IP 已存在于 TURN 服务器上，则会刷新，否则将创建。权限在 5 分钟后过期，除非刷新。

TURN 客户端仅在存在活跃分配时才允许调用 **CreatePermission** 方法。

如果权限创建成功，将调用事件 **OnTURNCreatePermission**。

```
oTURN->CreatePermission("80.147.23.157");
```

```
void OnTURNCreatePermission(TObject *Sender, const TsgcSocketConnection *aSocket,  
    const TsgcSTUN_Message *aMessage, const TsgcTURN_ResponseCreatePermission *aCreatePermission)  
{  
    DoLog("#Create Permission: " + aCreatePermission->IPAddresses->Text);  
}
```

TURN 客户端 | 发送指示

TURN 协议支持 2 种与对等方收发数据的机制，其中之一是 Send 和 Data 机制。

TURN 客户端可以使用 **SendIndication** 方法向服务器发送数据，由服务器中继给对端。TURN 客户端必须确保对等 IP 地址存在权限，才能发送指示。

SendIndication 方法的响应通过 **OnTURNDatIndication** 事件接收。

```
oTURN->SendIndication("80.147.23.157", 5000, "random data");

void OnTURNDatIndication(TObject *Sender, const TsgcSocketConnection *aSocket,
    const TsgcSTUN_Message *aMessage, const TsgcTURN_ResponseDataIndication *aDataIndication)
{
    DoLog("#Data Indication: [" + aDataIndication->PeerIP + ":" + IntToStr(aDataIndication->PeerPort) + "]" +
        sgcGetStringFromBytes(aDataIndication->Data));
}
```

TURN 客户端 | 通道

频道为 TURN 客户端和服务端提供了一种使用 `ChannelData` 消息发送应用程序数据的方式，其开销小于 `Send` 和 `Data` 指示。

在使用 `ChannelData` 消息在对端之间交换数据之前，TURN 客户端必须创建一个新频道，只需调用 `ChannelBind` 方法并传入对端 IP 地址和端口作为参数。

如果 TURN 服务器能够绑定新频道，TURN 客户端将收到成功响应 `OnTURNChannelBind` 事件。

```
oTURN->ChannelBind("80.147.23.157", 5000);

void OnTURNChannelBind(TObject *Sender, const TsgcSocketConnection *aSocket,
    const TsgcSTUN_Message *aMessage, const TsgcTURN_ResponseChannelBind *aChannelBind)
{
    DoLog("#Channel Bind: " + IntToStr(aChannelBind->Channel));
}
```

通道绑定持续 10 分钟，除非刷新。要刷新通道，只需再次调用 `ChannelBind` 方法。

当 TURN 客户端收到新的 `ChannelMessage` 时，调用 `OnTURNChannelData` 事件。

```
void OnTURNChannelData(TObject *Sender, const TsgcSocketConnection *aSocket,
    const TsgcTURNChannelData *aChannelData)
{
    DoLog("#Channel Data: [" + IntToStr(aChannelData->ChannelID) + "]" + " +
        sgcGetStringFromBytes(aChannelData->Data));
}
```

TsgcTURNServer

TURN 服务器支持长期认证、Allocation、Permissions、Channel Data 等功能。

简介

TsgcTURNServer 是实现 TURN 协议的服务器端组件，用于处理来自 TURN 客户端的请求。该组件继承自 **TsgcSTUNServer**，因此 TsgcTURNServer 上可使用所有方法和属性。

TURN 服务器支持长期认证、Allocation、Permissions、Channel Data 等功能。

基本用法

通常 TURN 服务器在 UDP 端口 3478 上运行并需要长期凭证，因此若要配置 TURN 服务器，请设置侦听端口（默认 3478）并启动服务器。

配置服务器

- Port：监听服务器端口，示例：3478
- 在 `TURNOptions.Authentication.LongTermCredentials` 中定义长期凭据属性。
- 处理 `OnSTUNRequestAuthorization` 以在 TURN 客户端向 TURN 服务器发送请求时设置密码。

将属性 **Active = True** 设置为启动 TURN 服务器。

```
TsgcTURNServer oTURN = new TsgcTURNServer();
oTURN->Port = 3478;
oTURN->TURNOptions->Authentication->Enabled = true;
oTURN->TURNOptions->Authentication->LongTermCredentials->Enabled = true;
oTURN->TURNOptions->Authentication->LongTermCredentials->Realm = "esegece.com";
oTURN->Active = true;
void OnSTUNRequestAuthorization(TObject *Sender, const TsgcSTUN_Message *aRequest,
    const string aUsername, const string aRealm, ref string Password)
{
    if ((aUsername == "user") && (aRealm == "esegece.com"))
    {
        Password = "password";
    }
}
```

参考

- 属性 — 组件上有 9 个可用属性。
- 方法 — 共 2 个公共方法。
- 事件 — 可处理 11 个事件。

指南

- 配置
 - [长期凭据](#)
- 分配
 - [Allocations](#)

TURN 服务器 | 长期凭据

通常，TURN 服务器配置为对 TURN 请求启用身份验证，对 STUN 请求不启用身份验证。

sgcWebSockets TURN 服务器支持长期凭据，您可以将 TsgcTURNServer 配置为仅允许携带长期凭据信息的请求。

要配置此功能，请访问 TURNOptions.Authorization 属性并启用它。

然后访问 LongTermCredentials 属性并启用它。默认情况下，此授权类型已配置了 Realm 字符串，并将默认 StaleNonce 值设为 10 分钟（= 600 秒）。

```
TsgcTURNServer oTURN = new TsgcTURNServer();
oTURN->STUNOptions->Authentication->Enabled = false;
oTURN->TURNOptions->Authentication->Enabled = true;
oTURN->TURNOptions->Authentication->LongTermCredentials->Enabled = true;
oTURN->TURNOptions->Authentication->LongTermCredentials->Realm = "sgcWebSockets";
oTURN->TURNOptions->Authentication->LongTermCredentials->StaleNonce = 600;
oTURN->Port = 3478;
oTURN->Active = true;

private void OnSTUNRequestAuthorization(TObject *Sender, const TsgcSTUN_Message *aRequest,
    const string aUsername, const string aRealm, ref string Password)
{
    if ((aUsername == "my-user") && (aRealm == "sgcWebSockets"))
    {
        Password = "my-password";
    }
}
```

TURN 服务器 | 分配

所有 TURN 操作都围绕分配展开，所有 TURN 消息都与一个分配相关联。一个分配由以下内容组成：

- 中继传输地址
- 五元组：客户端 IP 地址、客户端 IP 端口、服务器 IP 地址、服务器端口和传输协议。
- 身份验证信息。
- 每个中继传输地址的到期时间。
- 每个中继传输地址的权限列表。
- 每个中继传输地址的频道绑定列表。

当 TURN 客户端发送 **Allocate** 请求时，TURN 服务器处理该消息并尝试创建新的中继传输地址。默认情况下，如果有可用的 UDP 端口，将创建新的中继地址，但您可以使用 **OnTURNBeforeAllocate** 事件来拒绝新的分配请求。

```
void OnTURNBeforeAllocate(TObject *Sender, const TsgcSocketConnection *aSocket,
    const string aIP, Word aPort, ref bool Reject)
{
    if (your own rules) == false
    {
        Reject = false;
    }
}
```

如果流程继续，服务器将创建一个新的 Allocation，并调用事件 **OnTURNCreateAllocation**。此事件通过 **TsgcTURNAllocationItem** 类提供有关 Allocation 的信息。

```
void OnTURNCreateAllocation(TObject *Sender, const TsgcSocketConnection *aSocket,
    const TsgcTURNAllocationItem *Allocation)
{
    DoLog("New Allocation: " + Allocation->RelayIP + ":" + IntToStr(Allocation->RelayPort));
}
```

当分配过期或被删除（客户端发送生存期为零的刷新请求）时，触发 **OnTURNDeleteAllocation** 事件。

```
void OnTURNDeleteAllocation(TObject *Sender, const TsgcSocketConnection *aSocket,
    const TsgcTURNAllocationItem *Allocation)
{
    DoLog("Allocation Deleted: " + Allocation->RelayIP + ":" + IntToStr(Allocation->RelayPort));
}
```

ICE

交互式连接建立 (ICE) 协议用于 NAT 穿透。ICE 结合使用了多种方法，包括 Session Traversal Utility for NAT (STUN) 和 Traversal Using Relay NAT (TURN)。网络地址转换器 (NAT) 的存在给 VoIP 和 WebRTC 实现带来了问题。

组件

- **TsgcICEClient** : 实现 ICE 协议的客户端组件，允许您获取、交换和验证候选者。

TsgcICEClient

ICEOptions.

简介

TsgcICEClient 是实现 ICE 协议的客户端，允许您向 TURN 服务器发送分配请求。客户端需要 [TsgcTURNClient](#) 和一个 [TsgcWebSocketClient](#)。

配置

ICE 客户端具有以下属性

- **ICEOptions.CheckList.MaxCandidates**：客户端可处理的最大候选数，默认为 100。
- **ICEOptions.GatherCandidates.STUN**：默认为 true，将使用 STUN 协议获取候选者（反射地址）。
- **ICEOptions.GatherCandidates.TURN**：默认为 true，将使用 TURN 协议获取候选地址（中继地址）。

ICE 客户端需要一个 **TURN 客户端** 来收集候选，因此在调用方法 **GatherCandidates** 之前，请将 [TsgcTURNClient](#) 链接到 [TsgcICEClient](#) 的 **TURN** 属性。您可以获取有关如何配置 [TURN 客户端](#) 的更多信息。

```
TsgcICEClient *oICE = new TsgcICEClient();
TsgcTURNClient *oTURN = new TsgcTURNClient();
oTURN->Host = "www.esegece.com";
oTURN->Port = 3478;
oTURN->TURNOptions->Authentication->Credentials = stauLongTermCredential;
oTURN->TURNOptions->Authentication->Username = "sgc";
oTURN->TURNOptions->Authentication->Password = "secret";
oICE->GatherCandidates();
```

参考

- 属性 — 该组件提供 3 个属性。
- 方法 — 3 个公共方法。
- 事件 — 6 个可处理的事件。

指南

- 候选者
 - [收集候选者](#)
 - [配对候选者](#)

ICE | 收集候选

ICE 开始收集候选，通常会获取本地 IP 地址、使用 STUN 协议的反射地址以及使用 TURN 协议的中继地址。

要开始收集，请调用 **GatherCandidates** 方法，这将启动一个内部定时器，首先获取本地 IP 地址，然后连接到 STUN 服务器以获取反射 IP 地址，最后连接到 TURN 服务器以获取中继 IP 地址。

每次获取到新候选者时，事件 **OnICECandidate** 将被异步调用；如果在收集候选者时出现任何错误，则会触发事件 **OnICECandidateError**。

```
TsgcICEClient *oICE = new TsgcICEClient();
TsgcTURNClient *oTURN = new TsgcTURNClient();
oTURN->Host = "www.esegece.com";
oTURN->Port = 3478;
oTURN->TURNOptions->Authentication->Credentials = stauLongTermCredential;
oTURN->TURNOptions->Authentication->Username = "sgc";
oTURN->TURNOptions->Authentication->Password = "secret";
oICE->GatherCandidates();

void OnICECandidate(TObject *Sender, const TsgcICE_Candidate *aCandidate)
{
    DoLog("#Candidate" + aCandidate->AsString);
}
```

ICE | 候选对

获得候选（本地和远程）并设置 SDP 描述后，ICE 呼叫方客户端可以开始处理所有候选对，以找到可以交换数据的候选对。要启动此过程，请调用方法 **ProcessCandidates**。

方法 **ProcessCandidates** 评估所有候选对，发送 STUN 绑定数据包；如果从另一个对端收到 STUN 绑定数据包作为应答，则表示这两个对端之间可以建立连接，因此该对被提名。

当配对**成功**时，将异步触发事件 **OnICECandidatePairNominated**。如果配对出现**错误**或**超时后无法连接**，将触发事件 **OnICECandidatePairFailed**。

TsgcRTCPeerConnection

TsgcRTCPeerConnection 是一个客户端组件，允许您通过 UDP 使用 P2P 连接对等方。

简介

[*此组件目前正在开发中]

TsgcRTCPeerConnection 是一个客户端组件，允许您通过 UDP 使用 P2P 连接对等方。流程可分为 4 个步骤：

- 信令
- 正在连接
- 安全保护
- 通信中

为实现这些步骤，客户端使用以下协议：

- **WebSocket**：此协议用于信令，客户端交换会话描述协议以及本地、公共和中继 IP 地址。
- **UDP**：这是传输协议，客户端使用 UDP 在对等方之间发送/接收消息。
- **DTLS**：类似于 TLS，是一种加密规范，用于保护对端之间的消息，防止第三方读取/写入消息。
- **STUN**：用于获取公共 IP 地址的协议。
- **TURN**：当对端处于 NAT 后面时用于中继 IP 地址的协议。
- **ICE**：用于发现对等方之间可访问的 IP 地址和端口的协议。

信令

当客户端启动时，它不知道将与谁通信或通信内容是什么。信令使用 SDP（会话描述协议），其中包含以下详细信息：

- 对等方可到达的 IP 和端口。
- 用于保护通信安全的指纹证书。
- 用户名和密码。
- ...

信令使用 WebSocket 协议来交换数据，它通过子协议工作，在服务器端的 TsgcWSPServer_RTCPeerConnection 组件中实现。

TsgcRTCPeerConnection 内部已创建了一个 WebSocket 客户端，并附加了 TsgcWSPClient_RTCPeerConnection。

为了获取 IP 和端口，客户端使用 STUN/TURN 协议来获取此信息。因此，也需要 STUN/TURN 服务器。

链接：

- [RTCPeerConnection WebSocket 服务器](#)
- [RTCPeerConnection WebSocket 客户端](#)
- [RTCPeerConnection STUN TURN](#)
- [RTCPeerConnection 信令](#)

正在连接

一旦两个对等方知道候选者和 SDP，客户端将使用另一个称为 ICE 的标准协议。

ICE（交互式连接建立）允许在两个对等方之间建立连接。对等方可以在同一网络中，也可以位于 NAT 后面.....ICE 是无需中央服务器即可建立直连的解决方案。如果无法进行点对点连接，ICE 将使用 TURN 通过 TURN 服务器中继数据。

一旦 ICE 找到可以连接两个对等方的有效候选地址，下一步就是对通信进行加密。

链接：

- [RTCPeerConnection ICE](#)

安全保护

对端连接后，通信必须是安全的。这是通过 DTLS 实现的，DTLS 是一种用于保护 UDP 通信的加密协议。

DTLS 握手成功处理后，将使用另一种协议 SRTP（安全实时传输协议），目前尚未实现 SRTP。

链接：

- [RTCPeerConnection DTLS](#)

通信中

一旦两个对等方使用安全协议，通信将使用以下 2 种协议完成：

- **RTP**：实时传输协议：用于交换通过 SRTP 加密的媒体数据。
- **SCTP**：流控制传输协议，用于发送和接收经 dTLS 加密的 DataChannel 消息。

目前这些协议尚未实现，但您可以通过 UDP 上的 DTLS 发送/接收数据。

链接：

- [RTCPeerConnection 数据](#)

参考

- 属性 — 组件上有 2 个可用属性。
- 方法 — 3 个公共方法。
- 事件 — 14 个可处理的事件。

RTCPeerConnection | WebSocket 服务器

TsgcRTCPeerConnection 客户端需要 WebSocket 服务器进行信令。客户端使用 WebSocket 协议交换对等方的 SDP 及候选地址（IP 和端口），以便对等方进行通信。

要配置 **WebSocket 服务器**，您可以使用 sgcWebSockets 库中任何可用的 WebSocket 服务器 并附加一个 **TsgcWSPServer_RTCPeerConnection**，即 RTCPeerConnection 使用的子协议RTCPeerConnection。

```
TsgcWebSocketServer *oServer = new TsgcWebSocketServer();
TsgcWSPServer_RTCPeerConnection *oProtocol = new TsgcWSPServer_RTCPeerConnection();
oProtocol->Server = oServer;
oServer->Port = 8080;
oServer->Active = true;
```

每次有新的 WebSocket 客户端连接到 WebSocket 服务器时，服务器将检查是否有其他对等方在同一频道上监听，并相应地转发数据。

RTCPeerConnection | WebSocket 客户端

RTCPeerConnection 在内部创建一个具有自定义子协议的 WebSocket 客户端，用于与 WebSocket 服务器通信。在 RTCOptions.WebSocket 属性中，您可以找到定义 WebSocket 连接的各项值。

- **Host**：服务器的 DNS 或 IP 地址，示例：127.0.0.1 或 www.esegce.com。
- **Port**：WebSocket 服务器的监听端口。
- **TLS**：如果服务器使用安全连接，请启用它。
- **Channel**：用于在对端之间交换数据的频道名称（两个对端必须使用相同的频道名称，最大对端数为 2）。

RTCPeerConnection | STUN TURN

TsgcRTCPeerConnection 使用 STUN/TURN 协议获取公网 IP 地址和中继 IP 地址（如需要）。因此您需要一台 STUN/TURN 服务器来获取此信息。您可以从以下链接了解更多关于 STUN/TURN 服务器的信息：[TURN 服务器](#)。

一旦您的 STUN/TURN 服务器运行起来，您就可以在 TsgcRTCPeerConnection 的 RTCOptions.ICE 属性中配置 TURN 服务器属性。

- **Host**：TURN 服务器的 IP 地址或 DNS。示例：127.0.0.1 或 www.esegece.com。
- **Port**：通常是默认端口 3478。
- **Username**：如果 TURN 服务器使用长期凭据（默认），则为用户名。
- **Password**：TURN 服务器使用长期凭据（默认）时的密码。

RTCPeerConnection | 信令

TsgcRTCPeerConnection 配置好 RTCOptions 属性且服务器（WebSocket 和 STUN/TURN）运行之后，客户端可以开始收集候选者的过程。

客户端首先连接到 WebSocket 服务器，连接成功后发送本地 SDP。然后尝试获取本地和公共 IP 地址；为获取公共 IP 地址，将向 STUN 服务器发送绑定请求，以获取公共 IP 地址和 TURN 服务器的中继 IP 地址。每次获得新候选地址时，此信息将传递给 WebSocket 服务器，再转发给另一个对等方。

当 RTCPeerConnection 具有本地 SDP、远程 SDP 和候选者时，将开始检查每个候选者对，以确定它们之间是否可以连接。当某个候选者对成功连接时，它就是有效的候选者对，安全流程继续执行。

RTCPeerConnection | ICE

ICE（交互式连接建立）是用于连接两个对等方的协议，它确定两个对等方之间所有可能的路由，然后确保它们相互连接。这些路由也称为候选对，即本地与远程传输地址的配对。这些地址可以是本地 IP 地址、公网 IP 地址或中继传输地址。每个对等方收集所有可用地址，相互交换，然后尝试建立连接。

收集地址

在收集地址时可以调用以下事件

OnRTCLocalCandidate

找到新的本地候选者时调用该事件。

OnRTCRemoteCandidate

当 WebSocket 服务器向此对端发送远程候选时触发该事件。

OnRTCLocalDescription

当 TsgcRTCPeerConnection 需要本地 SDP 时调用此事件

OnRTCRemoteDescription

当 WebSocket 服务器将远程 SDP 发送给本对端时，触发此事件。

连通性测试

当对等方向另一对等方发送绑定请求以测试是否可以连接时，可能会调用以下事件

OnRTCCandidatePairNominated

当两个对等方均可使用该候选对连接时，将调用此事件。

OnRTCCandidatePairFailed

当对等方无法使用此候选对进行连接时，将调用此事件。

OnRTCConnect

当存在有效的候选对且未启用 DTLS 时调用此事件。
如果启用了 DTLS，在成功完成 DTLS 握手后将调用此事件。

RTCPeerConnection | DTLS

一旦存在有效的候选对（双方都能连接并在它们之间交换数据），就需要使连接变得安全。DTLS 是一种加密协议，用于加密数据，防止对所交换数据内容的检查或修改。

DTLS 需要 OpenSSL 库（OpenSSL 1.1 及以上）。

DTLS 的配置位于 `TsgcRTCPeerConnection` 的 `RTCOptions.DTLSOptions` 属性中。要启用 DTLS，请将 `RTCOptions.DTLS` 属性设置为 `True`。以下是主要属性：

RootCertFile：根证书文件的路径。

CertFile：证书文件的路径。

KeyFile：证书密钥文件的路径。

VerifyCertificate：如果必须验证证书，请启用此属性。

VerifyDepth：是一个整数属性，表示验证时允许的最大链接数

OpenSSL_Options：OpenSSL 库的配置。

APIVersion：允许定义将使用的 OpenSSL API。仅 openssl API 1.1 及以上版本支持 DTLS。

oslAPI_1_1：使用 OpenSSL API 1.1，需要我们定制的 Indy 库，并支持 OpenSSL 1.1.1 库（含 TLS 1.3 支持）。

oslAPI_3_0：使用 API 3.0 OpenSSL，需要我们的自定义 Indy 库，并允许使用 OpenSSL 3.0.0 库（支持 TLS 1.3）。

LibPath：在此处您可以配置 openssl 库的位置

oslpNone：这是默认值，OpenSSL 库应与二进制文件位于同一文件夹中，或位于已知路径中。

oslpDefaultFolder：自动设置所有 IDE 版本应查找 OpenSSL 库的路径。

oslpCustomFolder：如果选择此选项，请在属性 `LibPathCustom` 中定义完整路径。

LibPathCustom：当 `LibPath = oslpCustomFolder` 时，在此处定义 openssl 库所在的完整路径。

UnixSymLinks：启用或禁用在 Unix 系统下加载符号链接（默认启用，OSX64 除外）：

oslsSymLinksDefault：默认情况下启用，但在 OSX64 下除外（MacOS Monterey 之后，不带版本号尝试加载库会失败）。

oslsSymLinksLoadFirst：先加载符号链接，然后再尝试加载版本库。

oslsSymLinksLoad：在尝试加载版本库后加载符号链接。

oslsSymLinksDontLoad：不加载符号链接。

RTCPeerConnection | 数据

连接成功后，可以使用方法 **WriteData** 向另一个对等方发送消息。连接将使用基于 UDP 的 DTLS，如果可能，连接将是 P2P（如果 P2P 不可能，则使用中继地址）。

发送数据

使用 WriteData 方法向对端发送任何数据。您可以发送字符串或字节数组。

```
TsgcRTCPeerConnection *oRTCPeerConnection = new TsgcRTCPeerConnection();  
...  
oRTCPeerConnection->WriteData("Hello from sgcWebSockets!!!");
```

接收数据

每次 TsgcRTCPeerConnection 从对端接收到任何数据时，都会调用 OnRTCMessage 事件。

```
void OnRTCMessage(TObject *Sender, const TBytes aBytes)  
{  
    ShowMessage(TEncoding->UTF8->GetString(aBytes));  
}
```

Datasnap

默认情况下，DataSnap 使用 **TIdHttpWebBrokerBridge** 作为处理 HTTP 请求的服务器。您可以通过将此服务器替换为另一个支持更多协议且性能更佳的服务器来扩展 DataSnap 应用程序的功能，从而利用 HTTP/2 等最新协议（可提升服务器性能）以及 IOCP（允许处理更多连接）等特性。

服务器

服务器	主要功能	描述
TsgcWSHTTPWebBrokerBridgeServer	WebSocket 协议 HTTP 1.* 协议 XHR 协议 IOCP	基于 Indy 库，支持在同一端口上同时使用 WebSocket 和 HTTP 协议。也可以启用 IOCP。
TsgcWSHTTP2WebBrokerBridgeServer	WebSocket 协议 HTTP 1.* 协议 HTTP/2 协议 XHR 协议 IOCP	基于 Indy 库，在同一端口支持 WebSocket 和 HTTP/2 协议。也可启用 IOCP。
TsgcWSServer_HTTPAPI_WebBrokerBridge	WebSocket 协议 HTTP 1.* 协议 HTTP/2 协议 XHR 协议 IOCP	基于 HTTP.SYS Microsoft HTTP API，在同一端口上同时支持 WebSocket 和 HTTP/2 协议。默认使用 IOCP。建议用于最佳性能。

TsgcWSHTTPWebBrokerBridgeServer

TsgcWSHTTPWebBrokerBridgeServer 使用 **TIdHttpWebBrokerBridge** 作为服务器基础，适用于希望使用单一服务器同时处理 DataSnap、HTTP 和 WebSocket 连接的场景。

TsgcWSHTTPWebBrokerBridgeServer 继承自 **TsgcWebSocketHTTPServer**，因此您可以引用此服务器。

按照以下步骤将 **TIdHttpWebBrokerBridge** 替换为 **TsgcWSHTTPWebBrokerBridgeServer**：

1. 创建 **TsgcWSHTTPWebBrokerBridgeServer** 的新实例。
2. 将所有对 **TIdHttpWebBrokerBridge** 的调用替换为 **TsgcWSHTTPWebBrokerBridgeServer**。
3. 如需处理 WebSocket 连接，请参考 [TsgcWebSocketHTTPServer](#)。

配置

Datasnap 组件仅位于 **Source** 文件夹中，在编译后的文件夹中找不到，因为这些对象未包含在 **sgcWebSockets** 包中，所以必须在运行时创建。

只需将所需文件添加到您的项目中，或将路径设置到 **sgcWebSockets** 包的 **Source** 文件夹。所需文件：

- **sgcWebSocket_Server_WebBrokerBridge**

如果项目使用了 **IdHTTPWebBrokerBridge**，请将其更改为 **sgcIdHTTPWebBrokerBridge**（仅适用于 Enterprise Edition）。

事件

```
FServer = new TsgcWSHTTPWebBrokerBridgeServer();
FServer->OnCommandRequest = OnCommandRequestEvent;
FServer->OnCommandGet = OnCommandGetevent;

void OnCommandRequestEvent(TIdContext *AThread, TIdHTTPRequestInfo *ARequestInfo,
TIdHTTPResponseInfo *AResponseInfo, ref bool aHandled)
{
    if (ARequestInfo->Document == "/test.html")
    {
        aHandled = true;
    }
}

void OnCommandGetevent(TIdContext *AThread, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo
*AResponseInfo)
{
    if (ARequestInfo->Document == "/test.html")
    {
        AResponseInfo->ResponseNo = 200;
        AResponseInfo->ContentText = "hello all";
    }
}
```

负载均衡器

如果服务器位于 **TsgcWebSocketLoadBalancerServer** 后面，可能会遇到 CORS 问题，请使用以下代码来避免这些问题

```
void __fastcall TWebModule1::WebModuleBeforeDispatch(TObject *Sender, TWebRequest *Request, TWebResponse *Response)
{
    Response->SetCustomHeader("Access-Control-Allow-Origin", "*");
    if (Trim(Request->GetFieldByName("Access-Control-Request-Headers")) != "")
    {

```

```
        Response->SetCustomHeader("Access-Control-Allow-Headers", Request->GetFieldByName("Access-Control-Request-Headers"));
        Response->Handled = true;
    }
    if (FServerFunctionInvokerAction != nullptr)
    {
        FServerFunctionInvokerAction->Enabled = AllowServerFunctionInvoker;
    }
}
```

TsgcWSHTTP2WebBrokerBridgeServer

TsgcWSHTTP2WebBrokerBridgeServer 以后用了 HTTP/2 协议的 **TsgcWebSocketHTTPServer** 作为服务器基础，适用于希望使用单一服务器同时处理 DataSnap、HTTP/2 和 WebSocket 连接的场景。

TsgcWSHTTP2WebBrokerBridgeServer 继承自 **TsgcWebSocketHTTPServer**，您可以引用此服务器。

按照以下步骤将 **TIdHttpWebBrokerBridge** 替换为 **TsgcWSHTTP2WebBrokerBridgeServer**：

1. 创建 **TsgcWSHTTP2WebBrokerBridgeServer** 的新实例。
2. 将所有对 **TIdHttpWebBrokerBridge** 的调用替换为 **TsgcWSHTTP2WebBrokerBridgeServer**。
3. 如需处理 WebSocket 连接，请参考 [TsgcWebSocketHTTPServer](#)。

配置

Datasnap 组件仅位于 **Source** 文件夹中，在编译后的文件夹中找不到，因为这些对象未包含在 **sgcWebSockets** 包中，所以必须在运行时创建。

只需将所需文件添加到您的项目中，或将路径设置到 **sgcWebSockets** 包的 **Source** 文件夹。所需文件：

- **sgcWebSocket_Server_Base_WebBrokerBridge**
- **sgcWebSocket_Server_WebBrokerBridge_HTTP2**

如果项目使用了 **IdHTTPWebBrokerBridge**，请将其更改为 **sgcIdHTTPWebBrokerBridge**（仅适用于 Enterprise Edition）。

事件

```
FServer = new TsgcWSHTTP2WebBrokerBridgeServer();
FServer->OnCommandRequest = OnCommandRequestEvent;
FServer->OnCommandGet = OnCommandGetevent;

void OnCommandRequestEvent(TIdContext *AThread, TIdHTTPRequestInfo *ARequestInfo,
TIdHTTPResponseInfo *AResponseInfo, ref bool aHandled)
{
    if (ARequestInfo->Document == "/test.html")
    {
        aHandled = true;
    }
}

void OnCommandGetevent(TIdContext *AThread, TIdHTTPRequestInfo *ARequestInfo, TIdHTTPResponseInfo
*AResponseInfo)
{
    if (ARequestInfo->Document == "/test.html")
    {
        AResponseInfo->ResponseNo = 200;
        AResponseInfo->ContentText = "hello all";
    }
}
```


TsgcWSServer_HTTPAPI_WebBrokerBridge

TsgcWSServer_HTTPAPI_WebBrokerBridge 使用启用了 HTTP/2 协议的 **TsgcWebSocketServer_HTTPAPI** 作为服务器基础，适用于需要在单个服务器上处理 DataSnap、HTTP/2 和 WebSocket 连接的场景。

TsgcWSServer_HTTPAPI_WebBrokerBridge 继承自 **TsgcWebSocketServer_HTTPAPI**，因此您可以引用此服务器。

按照以下步骤将 **TIdHttpWebBrokerBridge** 替换为 **TsgcWSServer_HTTPAPI_WebBrokerBridge**：

1. 创建 **TsgcWSServer_HTTPAPI_WebBrokerBridge** 的新实例。
2. 将所有对 **TIdHttpWebBrokerBridge** 的调用替换为 **TsgcWSServer_HTTPAPI_WebBrokerBridge**。
3. 如需处理 WebSocket 连接，请参阅 [TsgcWSServer_HTTPAPI_WebBrokerBridge](#)。

配置

Datasnap 组件仅位于 **Source** 文件夹中，在编译后的文件夹中找不到，因为这些对象未包含在 **sgcWebSockets** 包中，所以必须在运行时创建。

只需将所需文件添加到您的项目中，或将路径设置到 **sgcWebSockets** 包的 **Source** 文件夹。所需文件：

- **sgcWebSocket_Server_Base_WebBrokerBridge**
- **sgcWebSocket_Server_HTTPAPI_WebBrokerBridge**

如果项目使用了 **IdHTTPWebBrokerBridge**，请将其更改为 **sgcIdHTTPWebBrokerBridge**（仅适用于 Enterprise Edition）。

事件

```
TsgcWSServer_HTTPAPI_WebBrokerBridge FServer = new TsgcWSServer_HTTPAPI_WebBrokerBridge();
FServer->OnCommandRequest += OnCommandRequestEvent;
FServer->OnMessage += OnWebSocketMessage;

void OnCommandRequestEvent(TsgcWSConnection_HTTPAPI *aConnection,
    const THttpRequest *aRequestInfo, ref THttpResponse *aResponseInfo, ref bool aHandled)
{
    if (ARequestInfo->Document == "/test.html")
    {
        AResponseInfo->ResponseNo = 200;
        AResponseInfo->ContentText = "... body ...";
        aHandled = true;
    }
}

void OnWebSocketMessage(TsgcWSConnection *aConnection, const string aText)
{
    aConnection->WriteData(aText);
}
```

OpenAPI

OpenAPI 3.0

The **OpenAPI 规范**，以前称为 **Swagger 规范**，是一种机器可读接口文件规范，用于描述、生成、使用和可视化 RESTful Web 服务。此前是 Swagger 框架的一部分，2016 年成为独立项目，由 Linux 基金会旗下的开源协作项目 OpenAPI Initiative 监管。Swagger 和其他一些工具可以根据接口文件生成代码、文档和测试用例。

基于 OpenAPI 接口文件实现的应用程序可以自动生成方法、参数和模型的文档。这有助于保持文档、客户端库和源代码的同步。

Pascal 解析器

sgcOpenAPI Generator 允许根据 OpenAPI 规范自动生成 API 客户端库（SDK 生成），支持以下 OpenAPI 规范：

- **OpenAPI 3.***
- **Swagger 2.***（自动从 2.0 转换为 3.0）
- **Swagger 1.***（自动从 1.0 转换为 3.0）

sgcOpenAPI 允许您 **自动** 根据 JSON/YAML OpenAPI 或 Swagger 以 **原生 Pascal 语言** 生成客户端 API 接口。目前支持从 Delphi 7 到最新 Delphi 版本。

sgcOpenAPI Generator 允许您从 OpenAPI / Swagger 规范创建文档文件。

了解更多关于 [OpenAPI Parser Pascal](#) 的信息。

OpenAPI 客户端

生成的客户端接口包含 OpenAPI 规范中定义的所有函数/方法。常量和枚举也会被创建。

支持以下 **Authentication** 方法：

- 基本认证
- [OAuth2 Code](#)（交互式）
- [OAuth2 凭据](#)（非交互式）
- [JWT](#)

阅读更多关于 [OpenAPI Client](#) 的内容。

OpenAPI Server

您还可以使用内置 OpenAPI 服务器在 Delphi 中托管 OpenAPI 3.0 服务。服务器根据您的规范自动路由请求，在 `/openapi.json` 提供规范，在 `/docs` 提供 Swagger UI，并开箱支持 CORS 和请求/正文验证。

定义 API 的两种方式：

- **规范优先**——在运行时加载现有的 OpenAPI 3.0 JSON/YAML 文件。
- **代码优先** – 使用 RTTI 属性注解 Delphi 类；规范自动生成（需要 Delphi XE7+）。

了解更多关于 [OpenAPI 服务器](#) 的信息。

API

以下 API 已编译并受支持：

- [Amazon AWS](#)
- [Google Cloud APIs](#)
- [Microsoft Azure](#)

- [其他 API](#)

OpenAPI | Parser Pascal

sgcOpenAPI 解析器读取 JSON 格式的 OpenAPI 3.0 规范，并自动生成 原生 Pascal 代码的 Delphi 客户端。

sgcOpenAPI 解析器兼容以下规范：

- OpenAPI 3.*
- Swagger 2.* (自动从 2.0 转换为 3.0)
- Swagger 1.* (自动从 1.0 转换为 3.0)

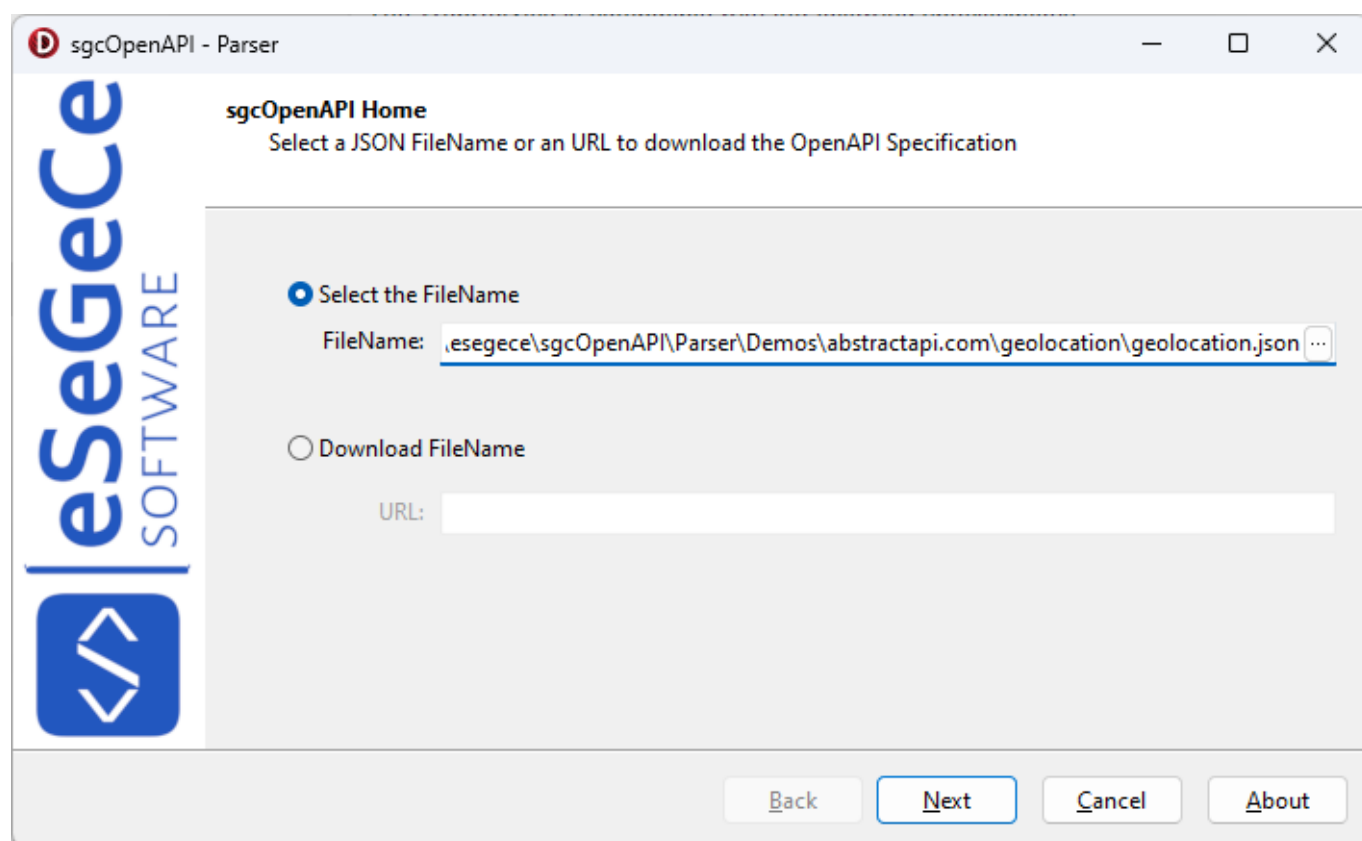
规范文件必须为 JSON 格式，如果规范为 YAML 格式，将自动转换。解析器允许导入拆分为多个模式的规范，如果提供 URL 将尝试下载规范，如果提供文件名将尝试从本地驱动器加载外部模式。

OpenAPI 解析器可执行文件可以处理命令行参数，也可以作为库使用。

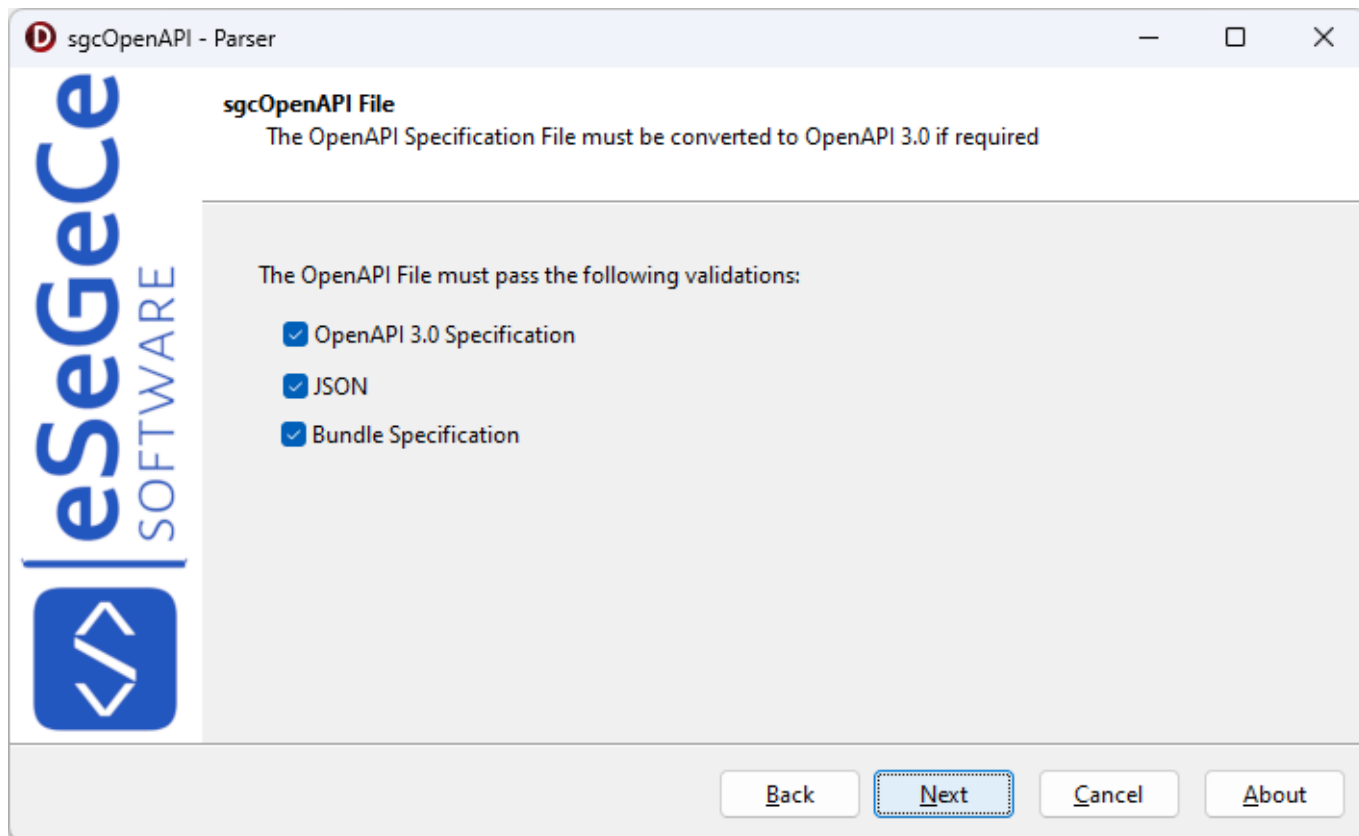
导入 OpenAPI 规范

第一步是导入 openAPI 规范。获得 JSON 格式的 openAPI 3.0 规范后，您可以使用我们的 **OpenAPI WebService** 生成所需的 Delphi 文件。请按照以下步骤操作：

- 执行 sgcOpenAPI 解析器并选择要导入的规范或要下载的 URL。



- 规范已验证，若与解析器兼容，您可继续执行后续步骤。如果规范使用的是 swagger 2.0 等旧版本，将自动进行转换。如果规范被拆分为多个 schema，解析器将尝试将其捆绑为单一规范。若转换文件时出现任何问题，将显示错误消息。



- 现在，您可以在解析文档之前自定义以下选项：
 - **ClassName**：这是主类名，默认以 "sgcOpenAPI_" 开头，并追加规范文件名。
 - **Namespace**：这是生成的 Pascal 文件名，默认与规范文件同名，扩展名为 ".pas"。
 - **Create Classes**：若勾选，将从规范文件创建类。例如：若某个请求需要发送 JSON 对象，且该 JSON 对象已在规范中定义，解析器将创建一个包含该 JSON 对象字段的类。
 - **Enable Classes**：若勾选，启用从规范文件生成的类与 JSON 对象配合使用（需要 Rad Studio XE7 或更高版本）。
 - **Documentation**：若勾选，解析器将为字段、类和方法添加注释。
 - **Authentication**：若存在，请选择任意一种身份验证方式。
 - **None**：API 不使用任何身份验证方法。
 - **Basic**：API 使用 BASIC 身份验证。
 - **Token**：需要以 HTTP 请求头的形式发送令牌，该令牌通过其他外部方法获取。
 - **OAuth2**：请求将使用 OAuth2 对 HTTP 请求进行身份验证。
 - **JWT**：请求将使用 JWT 对 HTTP 请求进行身份验证。

sgcOpenAPI Parser Options
Customize the options of the OpenAPI parser.

Classname:

Namespace:

Options: ☒ Create Classes ☒ Enable Classes ☒ Documentation

Authentication: ☒ None ☐ Basic ☐ Token ☐ OAuth2 ☐ JWT

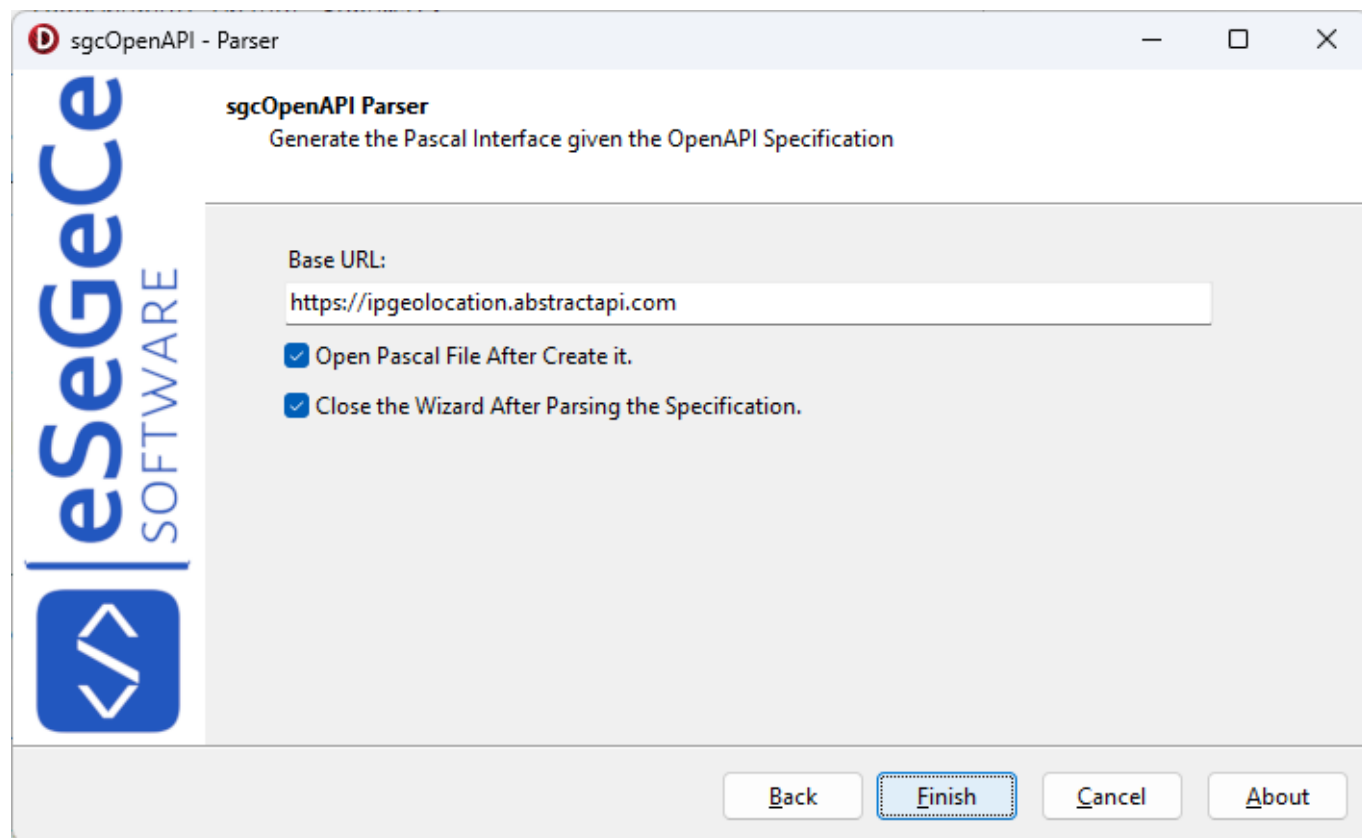
- 现在将显示一个包含已解析方法列表的表格作为信息。默认情况下，解析器使用 OperationId 作为所创建方法的名称，但如果 OperationId 未定义或值无效，您可以在此处使用摘要作为方法名称。

sgcOpenAPI Parsed Methods
Find below the list of parsed methods.

Import	Method	Description
☒	Retrieve_the_location_of_an_IP_address	Retrieve the location of an IP address

☐ Method Name From Summary

- 最后，验证 **Base URL** 是否具有正确的值，并在此处自定义生成文件是否在创建后自动打开，以及向导是否在生成文件后关闭。



- 按下**完成**以解析规范并生成 Pascal 文件。

示例

我将使用 abstractapi.com 所用的一个简单 OpenAPI 规范来检索某个 IP 地址的地理位置。

测试演示之前，您必须在 abstractapi.com 创建一个免费账户以获取 API 密钥。

<https://app.abstractapi.com/users/signup>

安装 [sgcOpenAPI 解析器安装程序](#)，打开 sgcOpenAPI.exe 并导入位于"Demos\abstractapi.com\geolocation"文件夹中的 OpenAPI 规范。导入并以名称"geolocation.pas"存储到演示文件夹后，打开演示项目，编译并运行。填写从 Abstractapi 网站获取的 API 密钥并按 Geolocation。

 sgcOpenAPI - Geolocation

API Key:

IP Address:

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\u0082\u2074",
    "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 | 附加属性

字典（也称为映射、哈希映射或关联数组）是一组键/值对。OpenAPI 允许您定义键为字符串的字典。要定义字典，请使用 `type: object`，并使用 `additionalProperties` 关键字指定键/值对中值的类型。例如，字符串到字符串的字典如下所示：

```
{
  "en": "English",
  "fr": "French"
}
```

OpenAPI 解析器使用 Embarcadero 的 JSON 类。在将 `TDictionary` 转换为 JSON 时，不是创建键/值对，而是创建 `TDictionary` 的所有内部对象，导致输出不正确。同样，尝试将 JSON 字符串转换为 `TDictionary` 对象时也会出现相同问题。

将 AdditionalProperties 转换为 JSON

OpenAPI 解析器将 `AdditionalProperties` 类创建为 `TsgcAdditionalProperties` 类型，因此如果需要将类转换为 JSON 字符串，必须创建一个继承自 `TsgcAdditionalProperties` 的新类，创建所需的所有字段，然后将其分配给属性类。

示例：给定一个具有 2 个属性的类，其中一个是附加属性，JSON 输出必须为

```
{"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;
```

创建一个继承自 `TsgcAdditionalProperties` 的新类，并创建 2 个属性

```
TCustomAdditionalProperties = class(TsgcAdditionalProperties)
private
  FKey1: string;
  FKey2: string;
public
  property Key1: string read FKey1 write FKey1;
  property Key2: string read FKey2 write FKey2;
end;
```

最后，将此类赋值给 `AdditionalProperties` 属性：

```
TMyClass *oClass = new TMyClass();
oClass->Property1 = "value1";
TCustomAdditionalProperties *oAdditionalProperties = new TCustomAdditionalProperties();
oAdditionalProperties->Key1 = "value1";
oAdditionalProperties->Key2 = "value2";
oClass->AdditionalProperties = oAdditionalProperties;
```

将 JSON 转换为 AdditionalProperties

由于 JSON 类的限制，OpenAPI Generator 创建了一个额外的方法，用于在 JSON 字符串解析后将 AdditionalProperties 加载到 TsgcAdditionalProperties.Dictionary 属性中。因此，要访问 AdditionalProperties 的内容，只需访问 TsgcAdditionalProperties.Dictionary 属性即可。

OpenAPI | 命令行

sgcOpenAPI 解析器包含一个命令行界面，可将 OpenAPI 规范转换为 Delphi（Pascal）客户端文件。解析器在生成输出之前对规范（OpenAPI 3 JSON + 捆绑模式）进行验证和规范化。

用法

基本语法

```
sgcOpenAPI.exe -i "c:\openapi.json" -o "c:\openapi.pas"
```

参数

所有参数均不区分大小写，短格式与长格式等效。

- **-i -input** : **[必填]** OpenAPI 规范的路径或下载 URL。如果输入为 YAML 或 Swagger 1/2 格式，将自动转换为 OpenAPI 3 JSON。
- **-o -output** : **[必填]** 要生成的输出 Pascal 文件（Delphi 客户端）。
- **-nc -nclasses** : **[可选]** 不为模式生成 Delphi 类（仅生成接口/客户端方法）。
- **-nj -njson** : **[可选]** 不为生成的类启用 JSON 序列化。
- **-nd -ndoc** : **[可选]** 不在输出中生成文档注释。
- **-m -method** : **[可选]** 生成方法的命名策略：
 - **0** : 使用 **OperationId**（默认）。
 - **1** : 使用 **Summary**。
 - **2** : 使用 **Endpoint** 路径。
- **-a -auth** : **[可选]** 在生成的客户端中启用的认证模式：
 - **0** : 无。
 - **1** : Basic。
 - **2** : Token。
 - **3** : OAuth2。
 - **4** : JWT。
- **-u -url** : **[可选]** 覆盖 OpenAPI 文件中检测到的基本 URL。
- **-l -log** : **[可选]** 解析时显示日志输出。
- **-user** : **[可选]** 许可证用户名（仅在许可证未激活时需要）。
- **-password** : **[可选]** 许可证密码（仅在许可证尚未激活时需要）。
- **-h -help** : 显示命令行帮助。

示例

1) 从本地文件生成 Delphi 客户端

```
sgcOpenAPI.exe -i "c:\specs\openapi.json" -o "c:\clients\orders.pas"
```

2) 从 URL 生成客户端，禁用类和文档

```
sgcOpenAPI.exe -i "https://api.example.com/openapi.json" -o "c:\clients\api.pas" -nc -nd
```

3) 使用摘要作为方法名并启用 OAuth2

```
sgcOpenAPI.exe -i "c:\specs\openapi.json" -o "c:\clients\auth.pas" -m 1 -a 3
```

4) 覆盖 Base URL 并显示日志输出

```
sgcOpenAPI.exe -i "c:\specs\openapi.json" -o "c:\clients\override.pas" -u "https://api.example.com/v2" -l
```

注意事项

- 如果输入规范无效或不是捆绑的 OpenAPI 3 JSON 文件，解析器将在生成之前尝试自动转换和捆绑它。
- 使用 URL 时，解析器会在验证并生成输出之前先下载规范。

OpenAPI | 库

OpenAPI 解析器可作为共享库（DLL）使用，这样您无需调用命令行工具即可从自己的应用程序生成 Pascal 客户端。该库导出了一小组封装相同解析器功能的函数，与 CLI 中可用的功能相同。

库函数

- **CreateOpenAPIPascalFile** - 使用默认选项从 OpenAPI 规范创建 Pascal 客户端文件。
- **CreateOpenAPIPascalFileEx** - 创建一个可完全控制解析器选项的 Pascal 客户端文件。
- **GetOpenAPILastErrors** - 返回解析器产生的最后一条错误消息。

CreateOpenAPIPascalFile

使用默认解析器选项（生成类、JSON 支持、文档、来自 **OperationId** 的方法名、无认证、无基础 URL 覆盖）生成 Pascal 客户端文件。

```
typedef bool (__stdcall *TCreateOpenAPIPascalFile)(const char* aInputFile, const char* aOutputFile);
```

参数

- **aInputFile**：要解析的 OpenAPI 规范的完整路径（或 URL）。如果文件为 YAML 或 Swagger 1/2 格式，解析器将在生成输出之前将其规范化为 OpenAPI 3 JSON 格式。
- **aOutputFile**：要创建的 Pascal 单元的完整路径。

返回值

如果文件生成成功，则返回 **True**，否则返回 **False**。如果失败，请调用 **GetOpenAPILastErrors** 获取错误消息。

示例

CreateOpenAPIPascalFileEx

生成包含对解析器选项进行显式控制的 Pascal 客户端文件。

```
typedef bool (__stdcall *TCreateOpenAPIPascalFileEx)(
    const char* aInputFile, const char* aOutputFile,
    bool aGenerateClasses, bool aGenerateJSON, bool aGenerateDocumentation,
    int aMethodName, int aAuthentication, const char* aBaseUrl, bool aEnableLog);
```

参数

- **aInputFile**：要解析的 OpenAPI 规范的完整路径（或 URL）。
- **aOutputFile**：要创建的 Pascal 单元的完整路径。
- **aGenerateClasses**：为 **True** 时，生成器会创建 schema 中定义的 Pascal 模型（类）。为 **False** 时，仅生成客户端接口。
- **aGenerateJSON**：为 **True** 时，在类中生成 JSON 序列化辅助方法；为 **False** 时，省略 JSON 支持。
- **aGenerateDocumentation**：当值为 **True** 时，将生成 OpenAPI 规范的文档文件。
- **aMethodName**：控制方法名称的生成方式：
 - **0** = OperationId（默认）。
 - **1** = Summary。
 - **2** = Endpoint。
- **aAuthentication**：向客户端添加认证辅助：

- **0** = 无。
- **1** = Basic。
- **2** = Token。
- **3** = OAuth2。
- **4** = JWT。
- **aBaseURL**：覆盖 OpenAPI 规范中的基础 URL。传入空字符串以保留原始值。
- **aEnableLog**：当值为 **True** 时，解析器将写入进度和验证消息，有助于故障排查。

返回值

如果文件生成成功，则返回 **True**，否则返回 **False**。如果失败，请调用 **GetOpenAPILastErrors** 获取错误消息。

示例

GetOpenAPILastErrors

返回描述解析器产生的最后一个错误（验证、转换或解析错误）的消息。在 **CreateOpenAPIPascalFile** 或 **CreateOpenAPIPascalFileEx** 返回 **False** 后调用此函数。

```
typedef const char* (__stdcall *TGetOpenAPILastErrors)();
```

返回值

返回错误数量

示例

OpenAPI | API

sgcOpenAPI 解析器允许使用自定义 DLL 获取已解析 Pascal 文件的更详细信息，只需创建一个包含 **sgcOpenAPI_CreateSection** 方法的新 DLL，并将库复制到解析器所在的同一文件夹中。每次创建新文件区段时，API 都会调用 CreateSection 方法，您可以根据需要修改内容。以下是一个简单的 Delphi 示例：

```
enum TsgcOpenAPIPascalSectionType {
    soapistHeaderStart, soapistHeaderEnums,
    soapistHeaderClasses, soapistHeaderMainClass, soapistHeaderClassesStructure,
    soapistHeaderMainInstance, soapistHeaderEnd, soapistImplementationStart,
    soapistImplementationClasses, soapistImplementationEnd
};

extern "C" __declspec(dllexport) void __stdcall sgcOpenAPI_CreateSection(
    int aType, wchar_t* &aValue)
{
    switch ((TsgcOpenAPIPascalSectionType)aType) {
        case soapistHeaderStart: break;
        case soapistHeaderEnums: break;
        case soapistHeaderClasses: break;
        case soapistHeaderMainClass: break;
        case soapistHeaderClassesStructure: break;
        case soapistHeaderMainInstance: break;
        case soapistHeaderEnd: break;
        case soapistImplementationStart: break;
        case soapistImplementationClasses: break;
        case soapistImplementationEnd: break;
    }
}
```

OpenAPI | 客户端

TsgcOpenAPI_Client 是一个非可视化组件，封装了根据 OpenAPI 规范发出 HTTP 请求的主要方法和属性。

使用 sgcOpenAPI 解析器创建的每个 OpenAPI 接口都有 **2 个方法**

1. **GetOpenAPIClient**：这是一个单例函数，返回主类的实例；若实例不存在，则自动创建。
2. **FreeOpenAPIClient**：如果主类已创建则释放它。

示例

使用 Abstractapi 检索 IP 地址的地理位置信息。

```
GetOpenAPIClient->Retrieve_the_location_of_an_IP_address("your api", "80.258.15.2");
```

身份验证

- **Basic**：使用用户名/密码作为 HTTP 标头进行身份验证。
 - **UserName**：用户名。
 - **Password**：密码。
- **OAuth2**：使用 OAuth2 进行身份验证，支持两种授权类型：
 - **auth2Code**：用于在大多数应用程序类型中执行身份验证和授权。该流程使应用能够安全获取 access_token、刷新令牌以及已登录用户的 ID 令牌。
 - **auth2ClientCredentials**：此授权类型通常用于必须在后台运行、无需与用户立即交互的服务器到服务器交互。
 - 了解更多关于 [OAuth2](#)。
- **JWT**：使用 JWT 进行身份验证。了解有关 [JWT](#) 的更多信息。
- **Token**：您可以传入 Bearer 令牌或任何其他自定义令牌值。

TLSOptions

允许配置如何使用 HTTP/1 协议连接到安全 SSL/TLS 服务器

ALPNProtocols：将发送到服务器的 ALPN 协议列表。

RootCertFile：根证书文件的路径。

CertFile：证书文件的路径。

KeyFile：证书密钥文件的路径。

Password：如果证书有密码保护，请在此处设置。

VerifyCertificate：如果必须验证证书，请启用此属性。

VerifyDepth：一个整数属性，表示对 X.509 证书执行验证时允许的最大链路数。

Version：默认使用 TLS 1.0，如果服务器需要更高的 TLS 版本，可在此处选择。

IOHandler：选择将用于 TLS 连接的库。

iohOpenSSL：使用 OpenSSL 库，是 Indy 组件的默认选项。需要为 win32/win64 部署 OpenSSL 库。

iohSChannel：使用由 Microsoft 为 Windows 实现的安全通道安全协议，无需部署 openssl 库。仅适用于 Windows 32/64 位。

OpenSSL_Options：OpenSSL 库的配置。

APIVersion：允许定义将使用哪个 OpenSSL API 版本。

oslAPI_1_0：使用 API 1.0 OpenSSL，这是 Indy 支持的最新版本

oslAPI_1_1：使用 OpenSSL API 1.1，需要我们定制的 Indy 库，并支持 OpenSSL 1.1.1 库（含 TLS 1.3 支持）。

oslAPI_3_0：使用 API 3.0 OpenSSL，需要我们的自定义 Indy 库，并允许使用 OpenSSL 3.0.0 库（支持 TLS 1.3）。

LibPath：在此处您可以配置 openssl 库的位置

oslpNone：这是默认值，OpenSSL 库应与二进制文件位于同一文件夹中，或位于已知路径中。

oslpDefaultFolder：自动设置所有 IDE 版本应查找 OpenSSL 库的路径。
oslpCustomFolder：如果选择此选项，请在属性 LibPathCustom 中定义完整路径。
LibPathCustom：当 LibPath = oslpCustomFolder 时，在此处定义 openssl 库所在的完整路径。
UnixSymLinks：启用或禁用 Unix 系统下加载符号链接（默认启用，OSX64 除外）：
 oslsSymLinksDefault：默认情况下启用，但在 OSX64 下除外（MacOS Monterey 之后，不带版本号尝试加载库会失败）。
 oslsSymLinksLoadFirst：先加载符号链接，然后再尝试加载版本库。
 oslsSymLinksLoad：在尝试加载版本库后加载符号链接。
 oslsSymLinksDontLoad：不加载符号链接。
SChannel_Options：允许您使用 Windows 证书存储中的证书。
CertHash：是证书哈希值。您可以在 PowerShell 中运行 dir 命令找到证书哈希值。
CipherList：在此设置将使用的密码套件（以":"分隔）。示例：CALG_AES_256:CALG_AES_128
CertStoreName：存储证书的存储名称。从以下选项中选择：
 scsnMY（默认值）
 scsnCA
 scsnRoot
 scsnTrust
CertStorePath：存储证书的存储路径。请从以下选项中选择：
 scspStoreCurrentUser（默认值）
 scspStoreLocalMachine

代理选项

使用此属性配置通过代理的连接。

Enabled：设为 true 以后用代理连接。
Host：代理服务器地址
Port：代理服务器端口
UserName/Password：连接代理的身份验证信息，仅在需要时填写。
ProxyType：支持以下代理类型：

- HTTP
- Socks4
- Socks4A
- Socks5

日志

若 Log 属性已启用，则将套接字消息保存到指定的日志文件，便于调试。

Log：如果希望将 HTTP 请求保存到文本文件，请启用此选项。
LogFileName：文件名的完整路径。

属性

可用于自定义 OpenAPI 客户端的其他属性：

EncodeBodyAsUTF8：若启用，JSON 正文将以 UTF8 编码（默认情况下为 false）。

事件

以下是使用 OpenAPI 客户端时可以处理的事件列表。

OnBeforeRequest

在调用 HTTP 请求之前调用此事件。允许自定义参数名称、标头、安全性.....以下是如何替换某些参数名称的示例。

```
void __fastcall OnBeforeRequestEvent(TObject *Sender, const TsgcOpenAPIRequest *aRequest)
{
    for (int i = 0; i < aRequest->Parameters->Count; i++)
    {
        TsgcOpenAPIParameter *oParameter = aRequest->Parameters->Items[i];
        if (oParameter->_Name == "meta-modified-from")
            oParameter->_Name = "eventDateTime-from";
        if (oParameter->_Name == "meta-modified-to")
            oParameter->_Name = "eventDateTime-to";
    }
}
```

OnUpload

当文件上传时调用此事件，您可以使用此事件了解上传进度。

OnDownload

当文件下载时调用此事件，您可以使用此事件了解下载进度。

OnSSLVerifyPeer

如果启用了证书验证，在此事件中您可以验证并决定是否接受服务器证书。

OnSSLGetHandler

此事件在创建 SSL 处理程序之前引发，您可以在这里创建自己的 SSL 处理程序（需要继承自 TIdServerIOHandlerSSLBase 或 TIdIOHandlerSSLBase）并设置所需的属性。

OnSSLAfterCreateHandler

如果未创建自定义 SSL 对象，则默认使用 OpenSSL 处理程序创建。您可以访问 SSL 处理程序属性并根据需要修改

OpenAPI | 服务器

sgcOpenAPI 服务器承载一个 HTTP 服务，其契约由 OpenAPI 3.0 规范描述。路由根据规范的 **paths** 部分自动构建，开箱即用提供 Swagger UI，并可根据每个操作声明的 JSON Schema 对传入请求进行验证。提供两个组件来承载服务器：

- **TsgcHTTPServer_OpenAPI**：独立 HTTP 服务器。自包含，内嵌 Indy HTTP 服务器，无需 sgcWebSockets。仅需要公开 OpenAPI 服务时推荐使用。
- **TsgcWSAPIServer_OpenAPI**：与 sgcWebSockets 兼容。附加到现有的 **TsgcWebSocketHTTPServer**，使同一服务器同时处理 WebSocket 和 HTTP/OpenAPI 流量。

两个组件共享相同的引擎、选项和事件。

TsgcHTTPServer_OpenAPI

在 **sgcHTTPServer_OpenAPI** 单元中定义的独立服务器。配置监听端口，加载 OpenAPI 规范并激活服务器。

TsgcWSAPIServer_OpenAPI

与 sgcWebSockets 兼容，定义于单元 **sgcWebSocket_Server_API_OpenAPI**。将现有的 **TsgcWebSocketHTTPServer** 分配给 **Server** 属性，使单个服务器实例能够同时分发 WebSocket 帧和 OpenAPI HTTP 请求。内部委托给独立组件使用的相同引擎，因此 API 接口完全相同。

内置端点

服务器自动提供以下端点（每个端点均可通过 **OpenAPIOptions.Endpoint** 禁用）：

/openapi.json：返回已加载的 OpenAPI 规范，由 **OpenAPIOptions.Endpoint.ServeSpec** 控制。

/docs：用于浏览和试用 API 的内嵌 Swagger UI，由 **OpenAPIOptions.Endpoint.ServeSwaggerUI** 控制。

OPTIONS *：CORS 预检响应，在 **OpenAPIOptions.CORS.Enabled** 为 True 时发送。

OpenAPIOptions

类型为 **TsgcOpenAPIServer_Options** 的顶层配置属性，包含三个子对象：

端点：内置端点及规范加载。

BasePath：添加到规范中声明的每个路由前的可选 URL 前缀。

SpecFile：如果已赋值，则当服务器变为 **Active** 时，该文件将被自动加载。

ServeSpec：若为 True（默认），加载的规范将通过 **/openapi.json** 提供服务。

ServeSwaggerUI：如果为 True（默认值），则 Swagger UI 在 **/docs** 处提供服务。

Validation：请求验证标志，全部默认为 False。

ValidateRequest：主开关，为整个请求启用验证。

ValidateRequestBody：根据规范中声明的 JSON Schema 验证请求正文。

ValidateQueryParams：根据已声明的模式验证查询参数。

ValidatePathParams：根据声明的模式验证路径参数。

ValidateRequired：当声明为必填的参数缺失时，拒绝该请求。

CORS：跨域支持。

Enabled：若为 True，则在每个响应上发送 CORS 标头，并自动响应 **OPTIONS** 预检请求。

AllowOrigins：**Access-Control-Allow-Origin** 的值（默认为 *）。

AllowHeaders：**Access-Control-Allow-Headers** 的值（默认 **Content-Type, Authorization**）。

AllowMethods：**Access-Control-Allow-Methods** 的值（默认为 **GET, POST, PUT, DELETE, PATCH, OPTIONS**）。

事件

以下是使用 OpenAPI 服务器时可以处理的事件列表。

OnBeforeRequest：在请求分发到 **OnRequest** 之前触发。将 **Accept** 参数设为 False 以用 403 响应拒绝请求。

OnRequest：主处理程序。检查 **aOperationId**，并使用 **aContext** 读取请求并写入响应。生成响应后，将 **Handled** 设置为 True。

OnAfterRequest：在用户处理程序返回后触发。适用于日志记录或指标收集。

OnAuthenticate：在需要认证时于分发前触发。将 **Authenticated** 设为 **False** 以返回 401 响应拒绝。

OnValidationError：当一条或多条验证规则失败时触发。检查 **aErrors**，并将 **Continue** 设为 **False** 以 400 响应拒绝该请求。

OnException：当未处理的异常从用户处理程序传播时触发。如果需要替换默认的 500 响应码，请调整 **aResponseCode**。

规范优先 vs 代码优先

服务器支持两种编写方式：

- **规范优先**：OpenAPI 3.0 JSON/YAML 文件是唯一可信来源；服务器通过解析该文件来发现路由。
- **代码优先**：带有 RTTI 注解的 Delphi 类是唯一的真实来源；OpenAPI 规范在启动时生成。需要 Delphi XE7 或更高版本。

OpenAPI | 服务器 | 规范优先

在 **Spec-First** 方法中，OpenAPI 3.0 规范（JSON 或 YAML）是唯一的事实来源：它描述了每个路径、操作、参数和 schema。服务器启动时会解析规范并自动构建路由表；您只需为每个操作提供业务逻辑。当 API 合约已经存在、与消费者共享，或在 Delphi 代码库之外维护时，推荐使用此 workflow。

加载规格

使用 **LoadFromFile** 从磁盘加载规范，或使用 **LoadFromString** 从内存字符串加载（例如从数据库检索或在运行时生成的规范）。

加载完成后，路由表将从规范的 **paths** 节构建。模板路径段（如 `/pets/{petId}`）会被自动识别，捕获的值通过 **aContext.PathParamAsXxx** 公开。

处理操作

服务器将每个传入请求分派到 **OnRequest** 事件，并传递规范中声明的 **operationId**。使用简单的 **if/else** 链（或字典查找）将操作路由到正确的处理程序。

读取请求数据

TsgcOpenAPIServerContext 对象为请求的每个部分公开了类型化访问器：

写入响应

上下文对象还提供了用于写入响应的辅助方法。最常用的辅助方法为 **RespondJSON** 和 **RespondError**；如需完全控制响应，可直接设置 **aContext.Response.Code** 和 **aContext.Response.ContentText**。

验证

当 **OpenAPIOptions.Validation** 下的验证标志启用时，服务器在调用 **OnRequest** 之前，将根据规范中声明的 JSON Schema 验证每个请求：

ValidateRequestBody：根据操作的 **requestBody** 所引用的 schema 校验请求体。

ValidateQueryParams：根据已声明的模式检查查询字符串参数。

ValidatePathParams：路径参数根据其声明的模式进行检查。

ValidateRequired：当声明为必填的参数缺失时，拒绝请求。

验证失败时，**OnValidationError** 事件携带错误列表触发。将 **Continue** 设为 **False** 以返回 400 响应拒绝请求，或将其保留为 **True**，让您的 **OnRequest** 处理程序来决定。

演示

一个完整的规范优先示例随附在 **Demos/30.Server/02.SpecFirst/** 下。

OpenAPI | 服务器 | Code-First

在代码优先方法中，Delphi 类是真相来源：方法、参数和 RTTI 属性描述 API 合约，OpenAPI 3.0 规范在启动时从中生成。生成后，规范通过 **LoadFromString** 加载到服务器中，并像来自文件一样构建路由表。
Code-First 需要 **Delphi XE7 或更高版本**，因为它依赖新的 RTTI 来检查属性。功能位于单元 **sgcHTTP_OpenAPI_Server_CodeFirst** 中，并由编译器指令 **{\$IFDEF DXE7}** 保护。

定义服务

使用属性标注 Delphi 类以描述 API。方法体为存根：实际逻辑位于服务器的 **OnRequest** 事件中；方法仅用于 RTTI 扫描器读取其签名。

可用属性

扫描器可识别以下属性。类级属性描述整个服务；方法级属性描述单个操作；参数级属性描述值的来源。

sgcServiceContract(name, description, version)：类级别。填充 OpenAPI info 块。

sgcRoute(path)：类级别的基路径，或相对于类基路径的方法级别路由。

sgcHttpGet, sgcHttpPost, sgcHttpPut, sgcHttpDelete, sgcHttpPatch, sgcHttpHead, sgcHttpOptions：方法级别。声明操作的 HTTP 动词。

sgcSummary(text)：方法级别。在规范和 Swagger UI 中显示的简短摘要。

sgcDescription(text)：方法级别。规格的详细描述。

sgcTag(name)：方法级别。在 Swagger UI 中将操作归组到同一标签下。

sgcResponse(code, description)：方法级。声明带有 HTTP 状态码和描述的响应。可重复使用以声明多个响应。

sgcFromPath, sgcFromQuery, sgcFromBody, sgcFromHeader：参数级别。声明参数的读取位置。

sgcRequired：参数级别。将参数标记为必填；当 **ValidateRequired** 启用时，缺少的参数将被验证报告。

生成和加载规格

使用 **TsgcOpenAPICodeFirstScanner** 从注解类生成 OpenAPI 3.0 规范，然后将结果加载到服务器中。

处理操作

扫描器从方法名称派生 **operationId**：方法 **ListTasks** 成为操作 **ListTasks**。在 **OnRequest** 中使用相同的 **operationId** 进行分发。

演示

完整的代码优先示例位于 **Demos/30.Server/01.CodeFirst/** 下。

OpenAPI | Amazon AWS

sgcOpenAPI Amazon AWS 客户端 (TsgcOpenAPI_Amazon_Client) 拥有自己的 OpenAPI 客户端，继承自 [TsgcOpenAPI_Client](#)。

此组件有一个名为 **AmazonOptions** 的属性，其中包含连接到 Amazon AWS 服务器所需的所有配置。

AmazonOptions

在 AmazonOptions 中，您可以定义所需的 **AccessKey** 和 **SecretKey**（必须事先从您的 Amazon 账户生成），以便对 Amazon AWS 服务器进行身份验证。

访问密钥授予对您资源的程序化访问权限。这意味着您必须像保管 AWS 账户根用户登录凭证一样谨慎保管访问密钥。

最佳实践是执行以下操作：

1. **创建 IAM 用户**，然后尽可能严格地定义该用户的权限。
2. 在该 IAM 用户下**创建访问密钥**。

获得凭据后，在以下属性中进行设置：

- AmazonOptions.AccessKey
- AmazonOptions.SecretKey

AmazonOptions.JSON 属性允许定义响应使用 JSON 格式还是 XML 格式。

AWS IAM 中的 IAM 角色、AWS IAM Identity Center (AWS Single Sign-On 的后继者) 中的用户以及联合用户拥有临时安全凭据。临时安全凭据在定义的时间段后或用户结束其会话时过期。您可以在以下属性中设置临时凭据的令牌：

- AmazonOptions.SessionToken

最常见的用途

- **配置**
 - [Amazon AWS 凭据](#)
- **API**
 - [Amazon AWS S3](#)

sgcOpenAPI AWS SDK

以下是当前可用 API 的列表。

- 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 (兼容 MongoDB)
- 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 信令频道
- 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 服务
- Amazon Translate
- Amazon WorkDocs
- Amazon WorkLink
- Amazon WorkMail
- Amazon WorkMail 消息流
- 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 成本与使用量报告服务
- 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 健康 API 和通知
- 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 安全隧道
- 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 节省计划
- 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
- 弹性负载均衡
- FinSpace 公共 API
- FinSpace 用户环境管理服务

- 防火墙管理服务
- Kafka 托管流式传输
- Kafka Connect 托管流
- Redshift Data API Service
- Route53 Recovery Cluster
- 架构
- Service Quotas
- 合成品

OpenAPI Amazon AWS | 凭据

AWS 根据您访问 AWS 的方式需要不同类型的安全凭证。例如，您需要用户名和密码登录 AWS 管理控制台，并需要访问密钥对 **AWS** 进行编程调用。

注意事项

- 请务必将以下内容保存在安全位置：与您 AWS 账户关联的电子邮件地址、AWS 账户 ID、root 用户密码以及账户访问密钥。如果您忘记或丢失了 root 用户密码，则需要通过与账户关联的电子邮件地址进行重置。如果您忘记或丢失了访问密钥，则必须登录账户以创建新密钥。
- 我们强烈建议您创建一个具有管理员权限的 IAM 用户用于日常 AWS 任务，并锁定根用户的密码和访问密钥。根用户仅用于限制为根用户专属的任务。
- 安全凭证是账户专属的。如果您可以访问多个 AWS 账户，则每个账户都有单独的凭证。
- 请勿将您的 AWS 凭据提供给第三方。

编程访问

您必须提供 AWS 访问密钥才能以编程方式调用 AWS。

创建访问密钥时，您会同时创建访问密钥 ID（例如 AKIAIOSFODNN7EXAMPLE）和秘密访问密钥（例如 wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY）。秘密访问密钥仅在创建时可供下载。如果您未下载秘密访问密钥或丢失了它，则必须创建新的密钥。

每个用户最多可分配两个访问密钥。拥有两个访问密钥在您想轮换它们时很有用。删除访问密钥后，它将永久消失且无法恢复。

以 root 用户身份登录时管理访问密钥

1. 以根用户身份登录 AWS 管理控制台。
2. 在导航栏右上角，选择您的账户名称或账号，然后选择 **我的安全凭证**。
3. 展开 **访问密钥（访问密钥 ID 和秘密访问密钥）** 部分。
4. 执行以下操作之一：
 - 要创建访问密钥，请选择 **创建新访问密钥**。如果您已有两个访问密钥，此按钮将被禁用，必须先删除一个访问密钥才能创建新密钥。出现提示时，选择 **显示访问密钥** 或 **下载密钥文件**。这是保存私有访问密钥的唯一机会。将私有访问密钥保存到安全位置后，选择 **关闭**。
 - 要停用访问密钥，请选择 **设为非活跃**。出现确认提示时，选择 **停用**。已停用的访问密钥仍计入您的两个访问密钥限制。
 - 要激活访问密钥，请选择 **设为活跃**。
 - 要在不再需要时删除访问密钥，请复制访问密钥 ID，然后选择 **删除**。删除访问密钥之前，必须先选择 **停用**。我们建议在永久删除之前确认访问密钥不再使用。要确认删除，请在文本输入框中粘贴访问密钥 ID，然后选择 **删除**。

以 IAM 用户身份登录时管理访问密钥

1. 以 IAM 用户身份登录 AWS 管理控制台。
2. 在右上角的导航栏中，选择您的用户名，然后选择 **我的安全凭证**。
3. 执行以下操作之一：
 - 要创建访问密钥，请选择 **Create access key**。如果您已有两个访问密钥，此按钮将被禁用，您必须删除一个访问密钥才能创建新密钥。出现提示时，选择 **Show secret access key** 或 **Download .csv file**。这是您保存私有访问密钥的唯一机会。将私有访问密钥保存到安全位置后，选择 **Close**。
 - 要停用访问密钥，请选择 **Make inactive**。出现确认提示时，选择 **Deactivate**。已停用的访问密钥仍计入您的两个访问密钥限额。
 - 要激活访问密钥，请选择 **Make active**。出现确认提示时，选择 **Make active**。
 - 当您不再需要访问密钥时，请复制访问密钥 ID，然后选择 **Delete**。这会停用访问密钥。我们建议您在永久删除之前验证访问密钥不再被使用。要确认删除，请将访问密钥 ID 粘贴到文本输入字段中，然后选择 **Delete**。

sgcOpenAPI 配置

一旦您拥有自己的 AWS 访问密钥，在向 Amazon AWS 服务器发出任何请求之前，必须在 OpenAPI Amazon 客户端中进行配置。

```
GetOpenAPIClient->AmazonOptions->AccessKey = "AKIAIOSFODNN7EXAMPLE";  
GetOpenAPIClient->AmazonOptions->SecretKey = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY";
```

OpenAPI Amazon AWS | S3

Amazon Simple Storage Service (Amazon S3) 是一项对象存储服务，提供业界领先的可扩展性、数据可用性、安全性和性能。各种规模和行业的客户都可以使用 Amazon S3 存储和保护任意数量的数据，适用于数据湖、网站、移动应用程序、备份与恢复、归档、企业应用程序、IoT 设备和大数据分析等各种用例。Amazon S3 提供管理功能，以便您优化、组织和配置对数据的访问权限，满足您的特定业务、组织和合规要求。

ListBuckets

```
GetOpenAPIClient->AmazonOptions->AccessKey = "AKIAIOSFODNN7EXAMPLE";  
GetOpenAPIClient->AmazonOptions->SecretKey = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY";  
GetOpenAPIClient->ListBuckets();
```

GetObject

```
GetOpenAPIClient->AmazonOptions->AccessKey = "AKIAIOSFODNN7EXAMPLE";  
GetOpenAPIClient->AmazonOptions->SecretKey = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY";  
GetOpenAPIClient->GetObject("bucket_name");
```

PutObject

```
GetOpenAPIClient->AmazonOptions->AccessKey = "AKIAIOSFODNN7EXAMPLE";  
GetOpenAPIClient->AmazonOptions->SecretKey = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY";  
GetOpenAPIClient->PutObject("bucket_name", "MyFile.txt", "payload");
```

OpenAPI | Google Cloud

sgcOpenAPI Google 客户端 (TsgcOpenAPI_Google_Client) 有其自己的 OpenAPI 客户端，继承自 [TsgcOpenAPI_Client](#)。

此组件有一个名为 **GoogleOptions** 的属性，其中包含连接到 Google Cloud 服务器所需的全部配置。

GoogleOptions

OpenAPI Google 客户端支持以下身份验证方式：

1. **OAuth2 Code**：是交互式流程，需要用户介入。
2. **JWT (服务账户)**：为非交互式方式，因此可作为服务运行。

身份验证在 `GoogleOptions.Authentication` 属性中配置，允许以下值：

- **oagaOAuth2**：交互式。
- **oagaJWT**：非交互式。您可以使用 **LoadSettingsFromFile** 方法从 JSON 文件导入设置。该方法将自动填充以下属性：
 - **ClientEmail**
 - **PrivateKeyId**
 - **PrivateKey**
 - **ServiceAccountOptions**：使用服务账号运行客户端时，需要以下属性：
 - **TokenURI**：默认情况下，值为 "https://oauth2.googleapis.com/token"，但使用 `LoadSettingsFromFile` 方法时该值会被更新。
 - **Subject**：使用全域委派时的电子邮件账号
 - **Scopes**：作用域列表。

最常见的用途

- **配置**
 - [Google Cloud OAuth2](#)
 - [Google Cloud 服务账户](#)
- **APIs**
 - [Google Cloud PubSub](#)
 - [Google Cloud Calendar](#)
 - [Google Cloud Drive](#)

sgcOpenAPI Google Cloud SDK

以下是当前可用 API 的列表。

- Abusive Experience Report API
- 加速移动页面 (AMP) URL API
- Access Approval API
- Access Context Manager API
- 广告交易所买家 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 发现服务
- 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 私有目录
- 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
- 数据管道 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

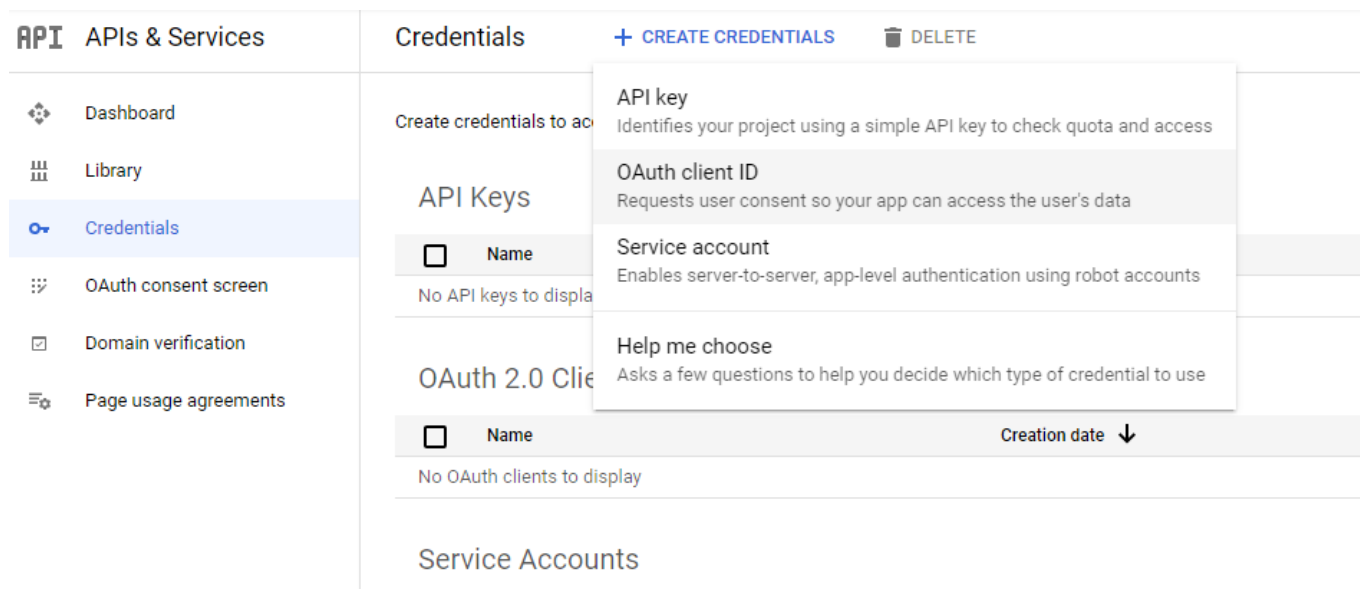
- Google Classroom API
- Google Cloud Data Catalog API
- Google Cloud Deploy API
- Google Cloud Memorystore for Redis API
- Google Cloud Support API
- Google Docs API
- Google Forms API
- Google Identity Toolkit API
- Google Keep API
- Google Mirror
- Google My Business API
- Google OAuth2 API
- Google Pay Passes API
- Google Play Android Developer API
- Google Play 自定义应用发布 API
- Google Play Developer Reporting API
- Google Play EMM API
- Google Play 游戏管理
- Google Play Game Services
- Google Play Game Services Publishing API
- Google Play Integrity API
- Google Search Console API
- Google Sheets API
- Google 网站验证 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 服务账号凭证 API
- Idea Hub API
- Identity and Access Management (IAM) API
- 索引 API
- 知识图谱搜索 API
- Kubernetes Engine API
- Library Agent API
- Local Services API
- Managed Service for Microsoft Active Directory API
- Manufacturer Center API
- 我的商家账户管理 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
- 副本池
- Resource Settings API
- Retail API
- Safe Browsing API
- SAS Portal API
- SAS Portal API (测试)
- 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
- 版本历史 API
- VM Migration API
- Web Fonts 开发者 API
- Web Risk API
- Web Security Scanner API
- Workflow Executions API
- Workflows API
- YouTube Analytics API
- YouTube Data API v3
- YouTube Reporting API

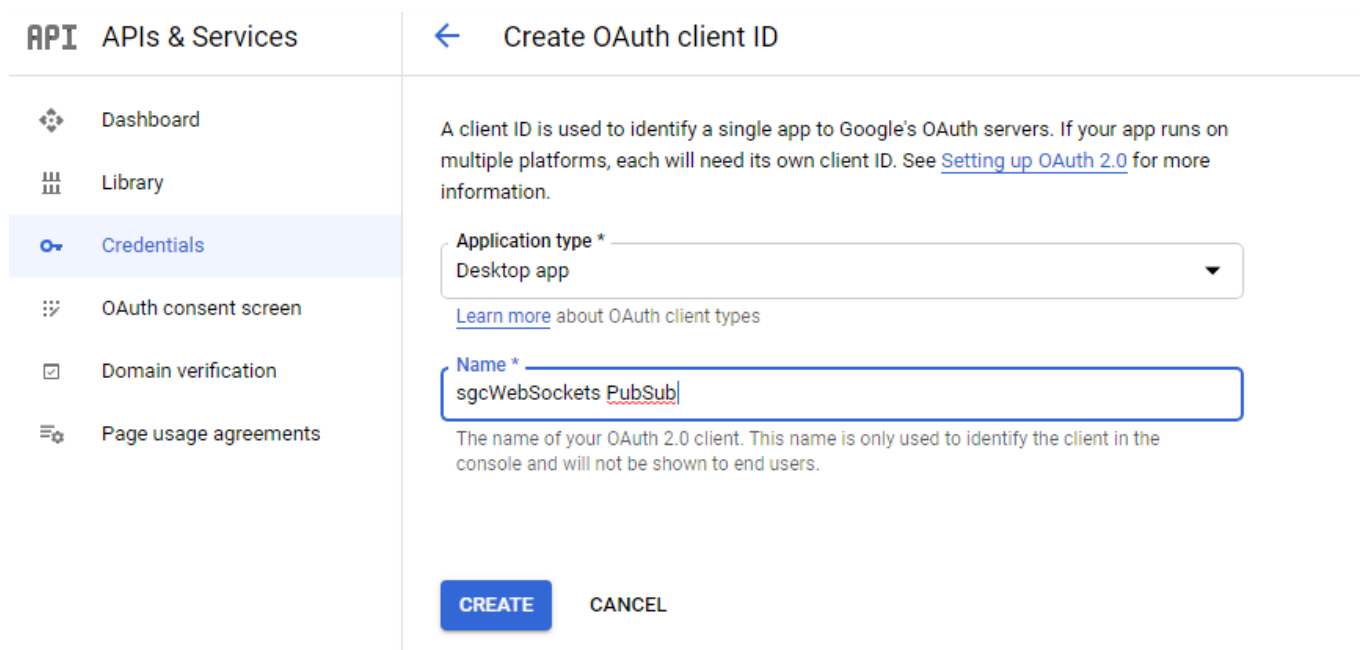
OpenAPI Google Cloud | OAuth2

要使用 OpenAPI Google Cloud 组件并通过 OAuth2 进行身份验证，首先必须从 Google Cloud 获取 OAuth2 密钥。请按照以下步骤获取 Google OAuth2 密钥，以及如何在我们的 PubSub 示例应用程序中进行配置。

首先登录您的 **Google Cloud** 账户，然后使用现有项目或创建一个新项目。完成后，前往凭据菜单，点击 **创建凭据** 按钮，选择 **OAuth 客户端 ID** 选项。



选择您的应用程序类型并设置描述性名称。



如果成功，您将获得您的 **Client Id** 和 **Client Secret**。

OAuth client created

The client ID and secret can always be accessed from Credentials in APIs & Services



OAuth is limited to 100 [sensitive scope logins](#) until the [OAuth consent screen](#) is verified. This may require a verification process that can take several days.

Your Client ID

843483347040-ehcskpfsp4180r1bdfoe6mc32e3ncmn0.apps.gc



Your Client Secret

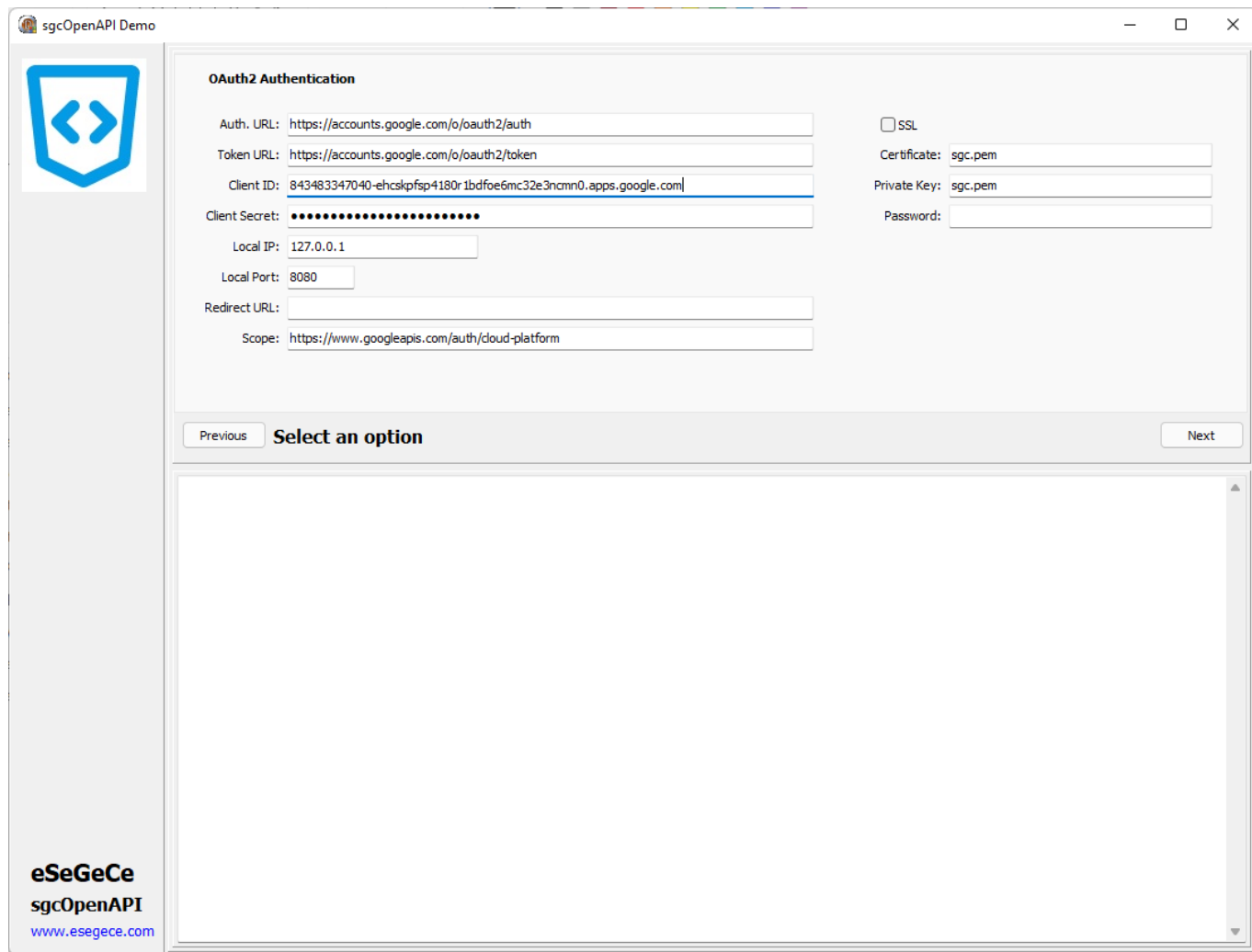
pvogD9reE0t9i1L6eR1jE60Z



OK

请勿与任何人共享您的 OAuth2 数据！

现在复制到 OpenAPI Google Cloud 示例



The image shows a screenshot of the 'sgcOpenAPI Demo' application window. The window has a title bar with standard Windows controls (minimize, maximize, close). On the left side, there is a vertical sidebar containing a blue shield logo with a white 'X' and the text 'eSeGeCe sgcOpenAPI www.egegece.com' at the bottom. The main area of the window is titled 'OAuth2 Authentication'. It contains several input fields for configuring OAuth2 settings. The fields are arranged in two columns. The left column includes: 'Auth. URL' (https://accounts.google.com/o/oauth2/auth), 'Token URL' (https://accounts.google.com/o/oauth2/token), 'Client ID' (843483347040-ehcspkfp4180r1bdf0e6mc32e3ncmn0.apps.google.com), 'Client Secret' (masked with dots), 'Local IP' (127.0.0.1), 'Local Port' (8080), 'Redirect URL' (empty), and 'Scope' (https://www.googleapis.com/auth/cloud-platform). The right column includes: an unchecked 'SSL' checkbox, 'Certificate' (sgc.pem), 'Private Key' (sgc.pem), and 'Password' (empty). Below the input fields, there is a 'Previous' button, a 'Select an option' label, and a 'Next' button. The bottom of the window is a large, empty white area with a scrollbar on the right.

sgcOpenAPI Demo

OAuth2 Authentication

Auth. URL:

Token URL:

Client ID:

Client Secret:

Local IP:

Local Port:

Redirect URL:

Scope:

☐ SSL

Certificate:

Private Key:

Password:

Select an option

eSeGeCe
sgcOpenAPI
www.egegece.com

了解更多关于 [OAuth2 配置](#) 的信息。

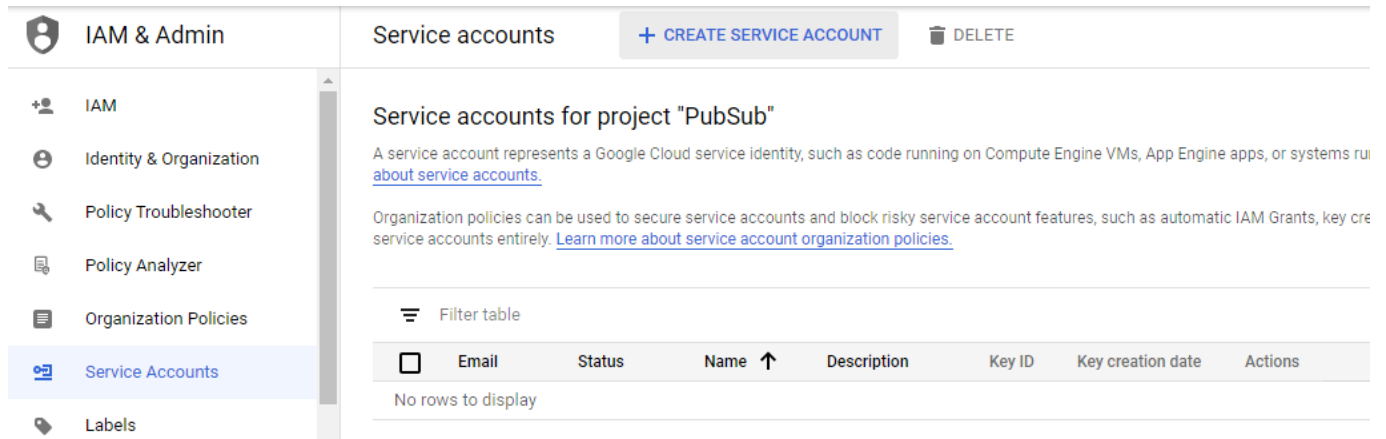
完成身份验证后，您可以先调用 **ClearOAuth2Token**（清除所有内部 OAuth2 令牌）方法进行**重新身份验证**，然后调用任何 OpenAPI 请求，此时将显示新的网络浏览器以重新针对 Google 服务器进行身份验证。

OpenAPI Google Cloud | 服务账号

若要使用 **OpenAPI Google Cloud** 组件并通过**服务账户**进行认证，首先必须从 Google Cloud 获取私钥证书。

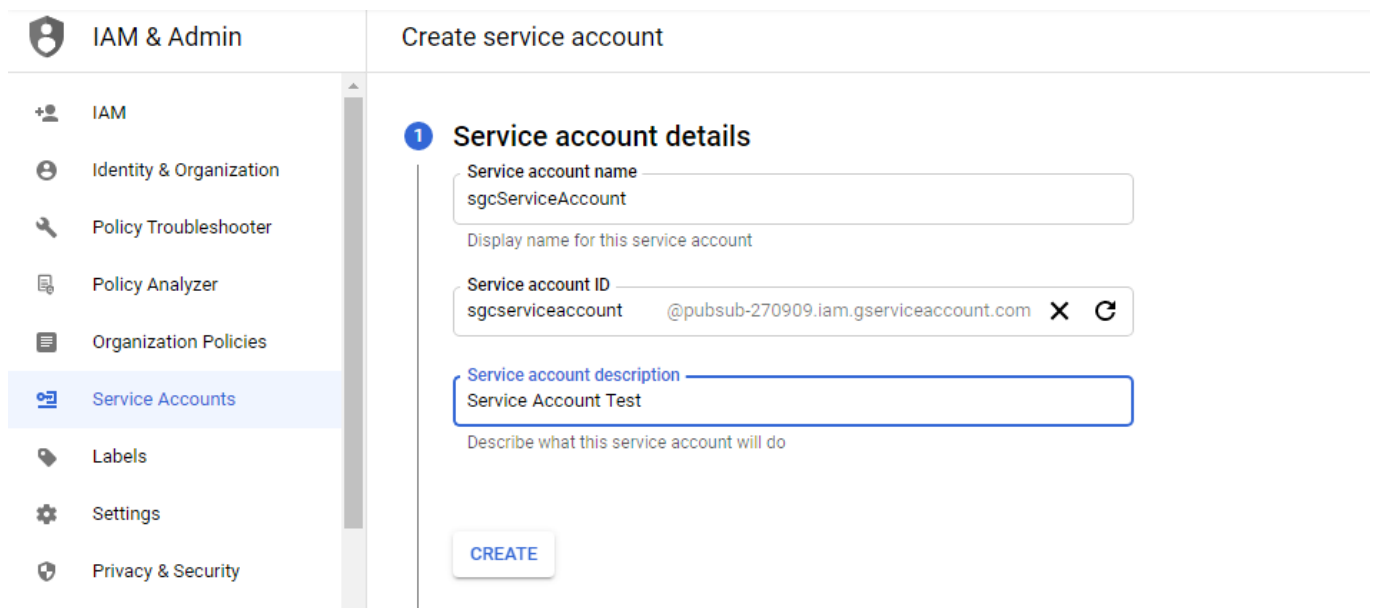
以下是获取 Google 私钥证书以及如何在我们的示例应用程序中进行配置的步骤。

首先**登录**您的 **Google Cloud** 账户，然后使用现有项目或创建一个新项目。



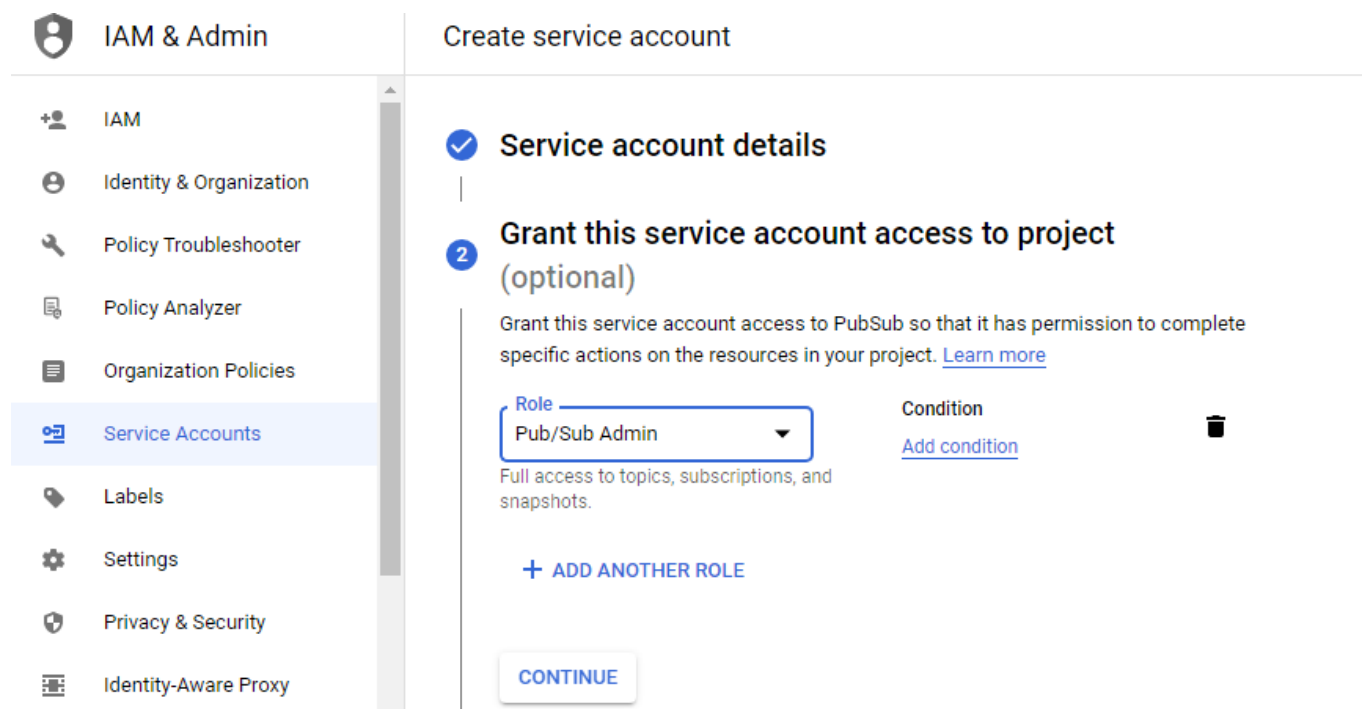
The screenshot shows the 'IAM & Admin' console with the 'Service accounts' tab selected. The page title is 'Service accounts for project "PubSub"'. Below the title, there is explanatory text about service accounts and a link to 'Learn more about service account organization policies'. A table with columns 'Email', 'Status', 'Name', 'Description', 'Key ID', 'Key creation date', and 'Actions' is shown, but it contains 'No rows to display'. A '+ CREATE SERVICE ACCOUNT' button is visible in the top right corner.

选择 **CREATE SERVICE ACCOUNT**，将显示一个新页面，您必须在其中设置服务账户名称和描述



The screenshot shows the 'Create service account' page. It has a section titled '1 Service account details' with three input fields: 'Service account name' (containing 'sgcServiceAccount'), 'Service account ID' (containing 'sgcserviceaccount' and '@pubsub-270909.iam.gserviceaccount.com'), and 'Service account description' (containing 'Service Account Test'). A 'CREATE' button is at the bottom.

然后至少选择一个角色，此处选择 PubSub Admin 以允许客户端发布和订阅主题，但您也可以选择其他权限较低的角色。



IAM & Admin

- IAM
- Identity & Organization
- Policy Troubleshooter
- Policy Analyzer
- Organization Policies
- Service Accounts**
- Labels
- Settings
- Privacy & Security
- Identity-Aware Proxy

Create service account

- Service account details
- Grant this service account access to project (optional)**

Grant this service account access to PubSub so that it has permission to complete specific actions on the resources in your project. [Learn more](#)

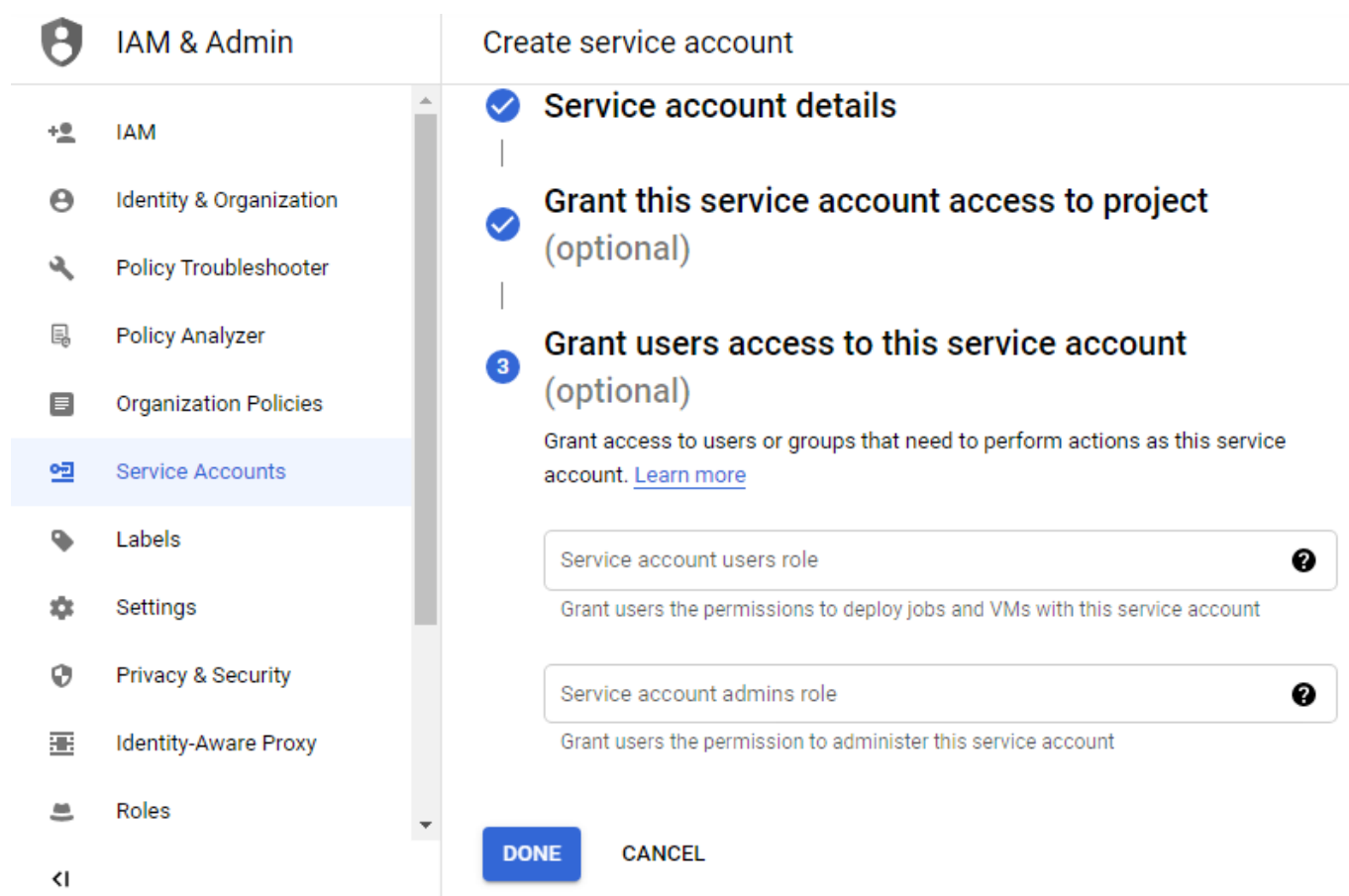
Role: **Pub/Sub Admin** Condition: [Add condition](#)

Full access to topics, subscriptions, and snapshots.

[+ ADD ANOTHER ROLE](#)

CONTINUE

按 CONTINUE 键，最终即可向其他用户授予访问权限



IAM & Admin

- IAM
- Identity & Organization
- Policy Troubleshooter
- Policy Analyzer
- Organization Policies
- Service Accounts**
- Labels
- Settings
- Privacy & Security
- Identity-Aware Proxy
- Roles

Create service account

- Service account details
- Grant this service account access to project (optional)
- Grant users access to this service account (optional)**

Grant access to users or groups that need to perform actions as this service account. [Learn more](#)

Service account users role ?
Grant users the permissions to deploy jobs and VMs with this service account

Service account admins role ?
Grant users the permission to administer this service account

DONE CANCEL

完成后按下 DONE，将显示新记录

IAM & Admin

IAM
Identity & Organization
Policy Troubleshooter
Policy Analyzer
Organization Policies
Service Accounts
Labels

Service accounts

+ CREATE SERVICE ACCOUNT

DELETE

Service accounts for project "PubSub"

A service account represents a Google Cloud service identity, such as code running on Compute Engine VMs, App Engine apps, or system [about service accounts](#).

Organization policies can be used to secure service accounts and block risky service account features, such as automatic IAM Grants, ke service accounts entirely. [Learn more about service account organization policies](#).

Filter table

☐	Email	Status	Name ↑	Description	Key ID
☐	sgcserviceaccount@pubsub-270909.iam.gserviceaccount.com	✓	sgcServiceAccount	Service Account Test	No keys

下一步是创建一个新密钥，因此在操作列中选择"创建密钥"选项。选择 JSON 以 JSON 格式下载配置，随即将创建一个新密钥。

Service accounts for project "PubSub"

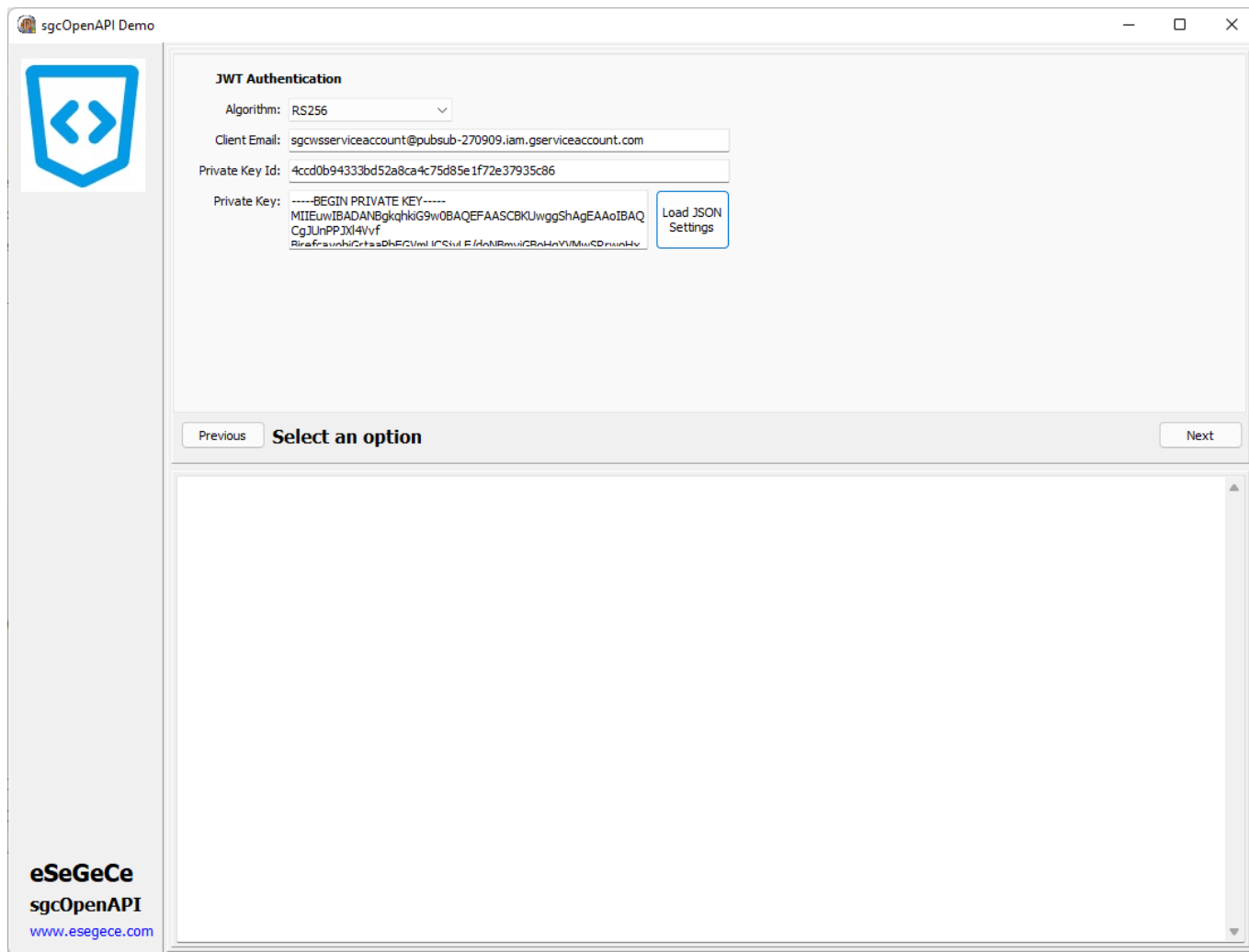
A service account represents a Google Cloud service identity, such as code running on Compute Engine VMs, App Engine apps, or systems running outside Google. [Learn more about service accounts](#).

Organization policies can be used to secure service accounts and block risky service account features, such as automatic IAM Grants, key creation/upload, or the creation of service accounts entirely. [Learn more about service account organization policies](#).

Filter table

☐	Email	Status	Name ↑	Description	Key ID	Key creation date	Actions
☐	sgcserviceaccount@pubsub-270909.iam.gserviceaccount.com	✓	sgcServiceAccount	Service Account Test	425a876f68b8b2a66f2fcc5b2dee8e918b0eb9ab	Dec 20, 2020	⋮

最后，您只需填入 Google 在 OpenAPI PubSub 客户端中提供的数据。您可以使用 **LoadSettingsFromFile** 加载配置 JSON 文件。



域范围委派

如果您拥有 Google Workspace 账户，组织的管理员可以授权应用程序代表 Google Workspace 域中的用户访问用户数据。例如，使用 **Google Calendar API** 为 Google Workspace 域中所有用户日历添加事件的应用程序，将使用服务账号代表用户访问 Google Calendar API。授权服务账号代表域中用户访问数据，有时称为向服务账号“委派全域权限”。

要向服务账户授予全域授权，Google Workspace 域的超级管理员必须完成以下步骤：

- 在您 Google Workspace 域的管理控制台中，前往**主菜单 > 安全 > 访问权限和数据控制 > API 控件**。
- 在**域范围委托**面板中，选择**管理域范围委托**。
- 单击**新增**。
- 在 **Client ID** 字段中，输入服务账户的 **Client ID**。您可以在服务账户页面找到服务账户的 Client ID。
- 在 **OAuth 作用域（逗号分隔）** 字段中，输入您的应用程序应被授予访问权限的作用域列表。例如，如果您的应用需要对 Google Drive API 和 Google Calendar API 进行域范围的完全访问，请输入：<https://www.googleapis.com/auth/drive>, <https://www.googleapis.com/auth/calendar>。
- 点击**授权**。

链接并授权工作区账户后，从 OpenAPI 客户端配置属性 `GoogleOptions.ServiceAccountOptions`：

- **Subject**：与服务账号关联的工作区电子邮件账号。示例：`youremail@domain.com`
- **作用域**：作用域列表。示例：`https://www.googleapis.com/auth/calendar`
- **TokenURI**：默认为 `https://oauth2.googleapis.com/token`。

OpenAPI Google Cloud | PubSub

Pub/Sub 允许服务以异步方式通信，延迟约为 100 毫秒量级。

Pub/Sub 用于流式分析和数据集成管道，以摄取和分发数据。它同样可作为面向服务集成的消息中间件或并行化任务的队列。

按主题列出项目（OAuth2）

```
GetOpenAPIClient->GoogleOptions->Authentication = oagaOAuth2;  
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->AuthURL = "https://accounts.google.com/o/oauth2";  
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->TokenURL = "https://accounts.google.com/o/oauth2/token";  
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/pubsub";  
GetOpenAPIClient->ListV1TopicsByProject("projects/pubsub-270909");
```

按主题列出项目（服务账号）

```
GetOpenAPIClient->GoogleOptions->Authentication = oagaJWT;  
GetOpenAPIClient->LoadSettingsFromFile("google.json");  
GetOpenAPIClient->ListV1TopicsByProject("projects/pubsub-270909");
```

OpenAPI Google Cloud Calendar

Calendar API 允许您将应用与 Google Calendar 集成，为您提供与用户互动的新方式。

按日历列出事件（OAuth2）

```
GetOpenAPIClient->GoogleOptions->Authentication = oagaOAuth2;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->AuthURL = "https://accounts.google.com/o/oauth2";
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->TokenURL = "https://accounts.google.com/o/oauth2/token";
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/calendar.events";
GetOpenAPIClient->ListCalendarsEventsByCalendarId("email@mydomain.com", true, "json", 0, "");
```

按日历列出事件（服务账号）

```
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, "");
```

插入日历事件

```
GetOpenAPIClient->InsertCalendarsEventsByCalendarId("email@mydomain.com", "{ \"summary\": \"Test Event\", \"description\": \"\" }", true, "json", 0, "");
```

OpenAPI Google Cloud | Drive

如果您想从 Google Drive 账户上传和/或下载文件，请使用单元 `sgcOpenAPI_googleapis.com_drive_helpers`。

上传文件

使用 `UploadFile` 方法将新文件上传到您的 Google Drive 账户。**OnUpload** 事件允许您了解上传进度。

```
void __fastcall UploadFileToGoogleDrive()
{
    TOpenDialog *oDialog = new TOpenDialog(NULL);
    try
    {
        if (oDialog->Execute())
        {
            GetOpenAPIClient()->OnUpload = OnUploadEvent;
            ShowMessage(GetOpenAPIClient()->UploadFile(oDialog->FileName));
        }
    }
    __finally
    {
        delete oDialog;
    }
}

void __fastcall OnUploadEvent(TObject *Sender, TsgcOpenAPIProgressState aState, __int64 aValue)
{
    switch (aState)
    {
        case oahpStart:
            ProgressBar1->Visible = true;
            ProgressBar1->Position = 0;
            ProgressBar1->Max = aValue;
            break;
        case oahpProgress:
            ProgressBar1->Visible = true;
            ProgressBar1->Position = aValue;
            break;
        case oahpEnd:
            ProgressBar1->Visible = false;
            ProgressBar1->Position = 0;
            ProgressBar1->Max = 0;
            break;
    }
    Application->ProcessMessages();
}
```

下载文件

使用方法 `DownloadFile` 从您的 Google Drive 账户下载文件。事件 **OnDownload** 允许您了解下载进度。

```
void __fastcall DownloadFileFromGoogleDrive()
{
    TSaveDialog *oSaveDialog = new TSaveDialog(NULL);
    try
    {
        if (oSaveDialog->Execute())
        {
            GetOpenAPIClient()->OnDownload = OnDownloadEvent;
            GetOpenAPIClient()->DownloadFile(txtDownloadFileId->Text, oSaveDialog->FileName);
        }
    }
    __finally
    {
        delete oSaveDialog;
    }
}

void __fastcall OnDownloadEvent(TObject *Sender, TsgcOpenAPIProgressState aState, __int64 aValue)
{
    switch (aState)
```

```

{
    case oahpStart:
        ProgressBar1->Visible = true;
        ProgressBar1->Position = 0;
        ProgressBar1->Max = aValue;
        break;
    case oahpProgress:
        ProgressBar1->Visible = true;
        ProgressBar1->Position = aValue;
        break;
    case oahpEnd:
        ProgressBar1->Visible = false;
        ProgressBar1->Position = 0;
        ProgressBar1->Max = 0;
        break;
}
Application->ProcessMessages();
}

```

列出文件夹

列出云端硬盘账户中的所有文件夹。使用 **ListFiles** 方法，并将参数 q 设为"application/vnd.google-apps.folder"。

```

GetOpenAPIClient().ListFiles("", true, "", false, false, 100, "", false, false, "", "", "", "", "", "", "", "", "", "", ""

```

OpenAPI | Microsoft

sgcOpenAPI Microsoft 客户端 (TsgcOpenAPI_Microsoft_Client) 拥有其自己的 OpenAPI 客户端，该客户端继承自 [TsgcOpenAPI_Client](#)。

此组件具有名为 **MicrosoftOptions** 的属性，其中包含连接到 Microsoft 服务器所需的所有配置。

MicrosoftOptions

OpenAPI Microsoft 客户端支持以下身份验证方式：

1. **OAuth2 Code**：是交互式流程，需要用户介入。
2. **OAuth2 凭证**：非交互式，因此可以作为服务运行。

身份验证在 `MicrosoftOptions.Authentication` 属性中配置，允许以下值：

- **oamaOAuth2Code**：交互式。
- **oamaOAuth2Credentials**：非交互式。

Microsoft 还需要以下属性：

- **TenantId**：这是标识您账户的值，可在您的 Microsoft/Azure 账户中找到。

最常见的用途

- **配置**
 - [Microsoft 获取租户](#)
 - [Microsoft 注册应用程序](#)
 - [Microsoft OAuth2 Code](#)
 - [Microsoft OAuth2 凭证](#)
- **APIs**
 - [Microsoft Graph](#)

sgcOpenAPI Microsoft APIs

以下是当前可用 API 的列表。

- AutoSuggest 客户端
- Computer Vision 客户端
- Custom Image Search Client
- Custom Search 客户端
- Custom Vision Prediction Client
- Custom Vision Training Client
- Entity Search Client
- 图片搜索客户端
- 本地搜索客户端
- News Search Client
- Partial Graph API
- 拼写检查客户端
- 视频搜索客户端
- Visual Search Client
- Web Search Client

sgcOpenAPI Azure APIs

以下是当前可用 API 的列表。

- API 客户端
- ACE Provisioning ManagementPartner
- ADHybridHealthService
- AdvisorManagementClient
- Anomaly Detector 客户端
- 异常检测器客户端
- ApiManagementClient
- AppConfigurationManagementClient
- Application Insights 数据平面
- ApplicationClient
- ApplicationInsightsManagementClient
- AppPlatformManagementClient
- AppServiceCertificateOrders API 客户端
- AppServiceEnvironments API 客户端
- AppServicePlans API 客户端
- 产物
- AttestationClient
- AuthorizationManagementClient
- AutomationManagement
- AutomationManagementClient
- Azure Action Groups
- Azure 活动日志警报
- Azure Addons 资源提供程序
- Azure 告警管理服务资源提供程序
- Azure Bot Service
- Azure CDN WebApplicationFirewallManagement
- Azure Data Catalog Resource Provider
- Azure Data Lake Storage
- Azure Data Migration Service Resource Provider
- Azure Dedicated HSM 资源提供程序
- Azure DevOps
- Azure Enterprise 知识图谱服务
- 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 模型管理服务
- Azure 机器学习工作区
- Azure Maps 资源提供程序
- 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 资源图查询
- Azure SQL 数据库
- Azure SQL Database API spec
- Azure SQL 数据库备份
- Azure SQL 数据库备份长期保留策略
- Azure SQL Database 数据掩码策略和规则
- Azure SQL 数据库灾难恢复配置
- Azure SQL 数据库导入/导出规范
- Azure SQL Database 复制链接
- Azure SQL Server API 规范
- Azure SQL Server 备份长期保留保管库

- Azure Stack Azure Bridge 客户端
- azureactivedirectory
- AzureAnalysisServices
- AzureBridgeAdminClient
- AzureDataManagementClient
- AzureDeploymentManager
- AzureDigitalTwinsManagementClient
- AzureStack Azure Bridge Client
- BackupManagementClient
- BatchAI
- BatchManagement
- BatchService
- BillingManagementClient
- BlockchainManagementClient
- BlueprintClient
- CdnManagementClient
- CertificateRegistrationProvider API Client
- Certificates API 客户端
- CognitiveServicesManagementClient
- CommerceManagementClient
- Compute Admin 客户端
- ComputeDiskAdminManagementClient
- ComputeManagementClient
- ComputeManagementConvenienceClient
- 计算机视觉
- ConsumptionManagementClient
- ContainerInstanceManagementClient
- ContainerRegistryManagementClient
- ContainerServiceClient
- 内容审核客户端
- 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 资源提供程序
- DomainRegistrationProvider API 客户端
- Domains API 客户端
- 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
- 访客诊断设置关联
- GuestConfiguration
- HanaManagementClient
- HDInsightJobManagementClient
- HDInsightManagementClient
- HealthcareApisClient
- HybridComputeManagementClient
- HybridDataManagementClient
- HyperDrive
- InfrastructureInsightsManagementClient
- Ink Recognizer 客户端
- InstanceMetadataClient
- IntuneResourceManagementClient
- iotDpsClient
- iotHubClient
- IoTSpacesClient
- KeyVaultClient
- KeyVaultManagementClient
- KustoManagementClient
- LogicAppsManagementClient
- LogicManagementClient
- LUIS Authoring Client
- LUIS Programmatic
- Machine Learning Compute Management 客户端
- 机器学习工作区管理客户端
- MaintenanceManagementClient
- ManagedLabsClient
- ManagedNetworkManagementClient
- ManagedServiceIdentityClient
- ManagedServicesClient
- 管理组
- ManagementLinkClient
- ManagementLockClient
- MariaDBManagementClient
- Marketplace RP Service
- MarketplaceOrdering.Agreements
- MediaServicesManagementClient
- Microsoft Insights
- Microsoft NetApp
- Microsoft Storage Sync
- Microsoft.ResourceHealth
- Microsoft.Support
- MicrosoftSerialConsoleClient
- 混合现实
- ML Team 账户管理客户端
- MonitorManagementClient
- MySQLManagementClient
- NetworkAdminManagementClient
- NetworkExperiments
- NetworkManagementClient
- NotificationHubsManagementClient
- PeeringManagementClient
- Personalizer Client
- PolicyClient
- PolicyEventsClient
- PolicyMetadataClient
- PolicyStatesClient

- PolicyTrackedResourcesClient
- 门户
- PostgreSQLManagementClient
- Power BI Embedded Management Client
- PowerBIDedicated
- PrivateDnsManagementClient
- Provider API 客户端
- QnAMaker Client
- QnAMaker Runtime Client
- Recommendations API 客户端
- RecoveryServicesBackupClient
- RecoveryServicesClient
- RedisManagementClient
- 中继
- RemediationsClient
- ResourceHealthMetadata API Client
- ResourceManagementClient
- 运行历史 API
- RunCommandsClient
- SchedulerManagementClient
- SeaBreezeManagementClient
- SearchIndexClient
- SearchManagementClient
- SearchServiceClient
- 安全中心
- 安全洞察
- ServerManagement
- Service Fabric Client APIs
- 服务地图
- 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 客户端
- TrafficManagerManagementClient
- 更新管理
- UpdateAdminClient
- UsageManagementClient
- VirtualMachineImageTemplate
- VirtualWANAsAServiceManagementClient
- Visual Studio Projects Resource Provider Client
- Visual Studio 资源提供程序客户端
- VM Insights 载入
- VMwareCloudSimple
- WebApplicationFirewallManagement
- WebApps API 客户端
- WebSite Management Client
- windowsesu
- WorkbookClient
- Workload Monitor

OpenAPI Microsoft | 租户

要构建使用 Microsoft 身份平台进行身份和访问管理的应用程序，您需要访问 Azure Active Directory (Azure AD) 租户。您在 Azure AD 租户中注册和管理应用程序、配置其对 Microsoft 365 和其他 Web API 中数据的访问，以及启用条件访问等功能。

租户代表一个组织。它是 Azure AD 的专用实例，组织或应用程序开发人员在与 Microsoft 建立关系之初获得。这种关系可能始于注册 Azure、Microsoft Intune 或 Microsoft 365 等服务。

每个 Azure AD 租户都是独立且与其他 Azure AD 租户分离的。它有其自己的工作、学校身份、消费者身份（如果是 Azure AD B2C 租户）以及应用注册。租户内的应用注册可以仅允许来自租户内账户或所有租户的身份验证。

使用现有的 Azure AD 租户

要检查租户：

1. 登录 Azure 门户。使用您将用于管理应用程序的账户。
2. 检查右上角。如果您有租户，您将自动登录。您会在账户名正下方看到租户名称。

如果您的账户没有关联的租户，您将在账户名下看到一个 GUID。在创建 Azure AD 租户之前，您将无法执行注册应用等操作。

创建新的 Azure AD 租户

您需要提供以下信息来创建新租户：

- **组织名称**
- **初始域** - 初始域 <domainname>.onmicrosoft.com 无法编辑或删除。您可以稍后添加自定义域名。
- **国家或地区**

OpenAPI Microsoft | 注册应用程序

注册您的应用程序将在您的应用与 Microsoft 标识平台之间建立信任关系。该信任关系是单向的：您的应用信任 Microsoft 标识平台，而非反向。

按照以下步骤创建应用程序注册：

1. 登录 Azure 门户。
2. 如果您有多个租户的访问权限，请使用顶部菜单中的 **目录 + 订阅** 筛选器 切换到您要注册应用程序的租户。
3. 搜索并选择 **Azure Active Directory**。
4. 在 **管理** 下，选择 **应用注册** > **新注册**。
5. 为您的应用程序输入显示 **名称**。您的应用程序用户在使用该应用时（例如在登录期间）可能会看到此显示名称。您可以随时更改显示名称，且多个应用注册可以共享同一个名称。在身份平台中，用于唯一标识您的应用的是应用注册自动生成的应用程序（客户端）ID，而非显示名称。
6. 指定谁可以使用该应用程序，有时也称为其 **登录受众**。
7. 选择 **注册** 以完成初始应用注册

Register an application - Microsoft x +

← → ↻ <https://portal.azure.com> ☆ 📁 👤 ...

Microsoft Azure Search resources, services, and docs (G+/) [Icons] meganb@contoso.com CONTOSO AD (DEV)

Home > Contoso AD (dev) >

Register an application ×

* Name

The user-facing display name for this application (this can be changed later).

Supported account types

Who can use this application or access this API?

☒ Accounts in this organizational directory only (Contoso AD (dev) only - Single tenant)

☐ Accounts in any organizational directory (Any Azure AD directory - Multitenant)

☐ Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)

☐ Personal Microsoft accounts only

[Help me choose...](#)

Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web ▼ e.g. <https://myapp.com/auth>

By proceeding, you agree to the [Microsoft Platform Policies](#) [↗](#)

[Register](#)

添加重定向 URI

重定向 URI 是 Microsoft 身份平台在身份验证后重定向用户客户端并发送安全令牌的位置。

例如，在生产 Web 应用程序中，重定向 URI 通常是应用程序运行的公共端点，例如 `https://contoso.com/auth-response`。在开发过程中，通常还会添加应用程序在本地运行的端点，例如 `https://127.0.0.1/auth-response` 或 `http://localhost/auth-response`。

此 **RedirectURI** 将在稍后用于配置 **sgcOpenAPI Microsoft Client**。

添加凭据

凭据由访问 Web API 的保密客户端应用程序使用。保密客户端的示例包括 Web 应用、其他 Web API，或服务类型 and 守护进程类型的应用程序。凭据允许您的应用程序以自身身份进行身份验证，运行时无需用户交互。

您可以将证书和客户端密钥（字符串）都作为凭据添加到机密客户端应用程序注册中。

Microsoft Azure Search resources, services, and docs (G+)

Home > Fourth Coffee > Contoso App 1

Contoso App 1 | Certificates & secrets

Search (Ctrl+/) Got feedback?

- Overview
- Quickstart
- Integration assistant

Manage

- Branding
- Authentication
- Certificates & secrets**
- Token configuration
- API permissions
- Expose an API
- App roles
- Owners
- Roles and administrators | Preview
- Manifest

Support + Troubleshooting

- Troubleshooting
- New support request

Credentials enable confidential applications to identify themselves to the authentication service when receiving tokens at a web addressable location (using an HTTPS scheme). For a higher level of assurance, we recommend using a certificate (instead of a client secret) as a credential.

Application registration certificates, secrets and federated credentials can be found in the tabs below.

Certificates (0) **Client secrets (0)** Federated credentials (0)

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

+ New client secret

Description	Expires	Value	Secret ID
No client secrets have been created for this application.			

添加客户端密钥

有时也称为 *应用程序密码*，客户端密钥是您的应用程序可以用来替代证书来标识自身的字符串值。

与证书凭据相比，客户端密钥被认为安全性较低。应用程序开发者有时在本地应用开发期间使用客户端密钥，因为它们使用简便。但是，对于在生产环境中运行的任何应用程序，您应该使用证书凭据。

1. 在 Azure 门户中，在 **应用注册** 中选择您的应用程序。
2. 选择 **证书和机密** > **客户端机密** > **新客户端机密**。

3. 为您的客户端密钥添加描述。
4. 为密钥选择到期时间或指定自定义有效期。
 - 客户端密钥的有效期限限制为两年（24 个月）或更短。您不能指定超过 24 个月的自定义有效期。
 - Microsoft 建议将到期值设置为 12 个月以内。
5. 选择 **添加**。
6. *请记录密钥值* 以供在您的客户端应用程序代码中使用。离开此页面后，此密钥值 *将不再显示*。

OpenAPI Microsoft | OAuth2 代码

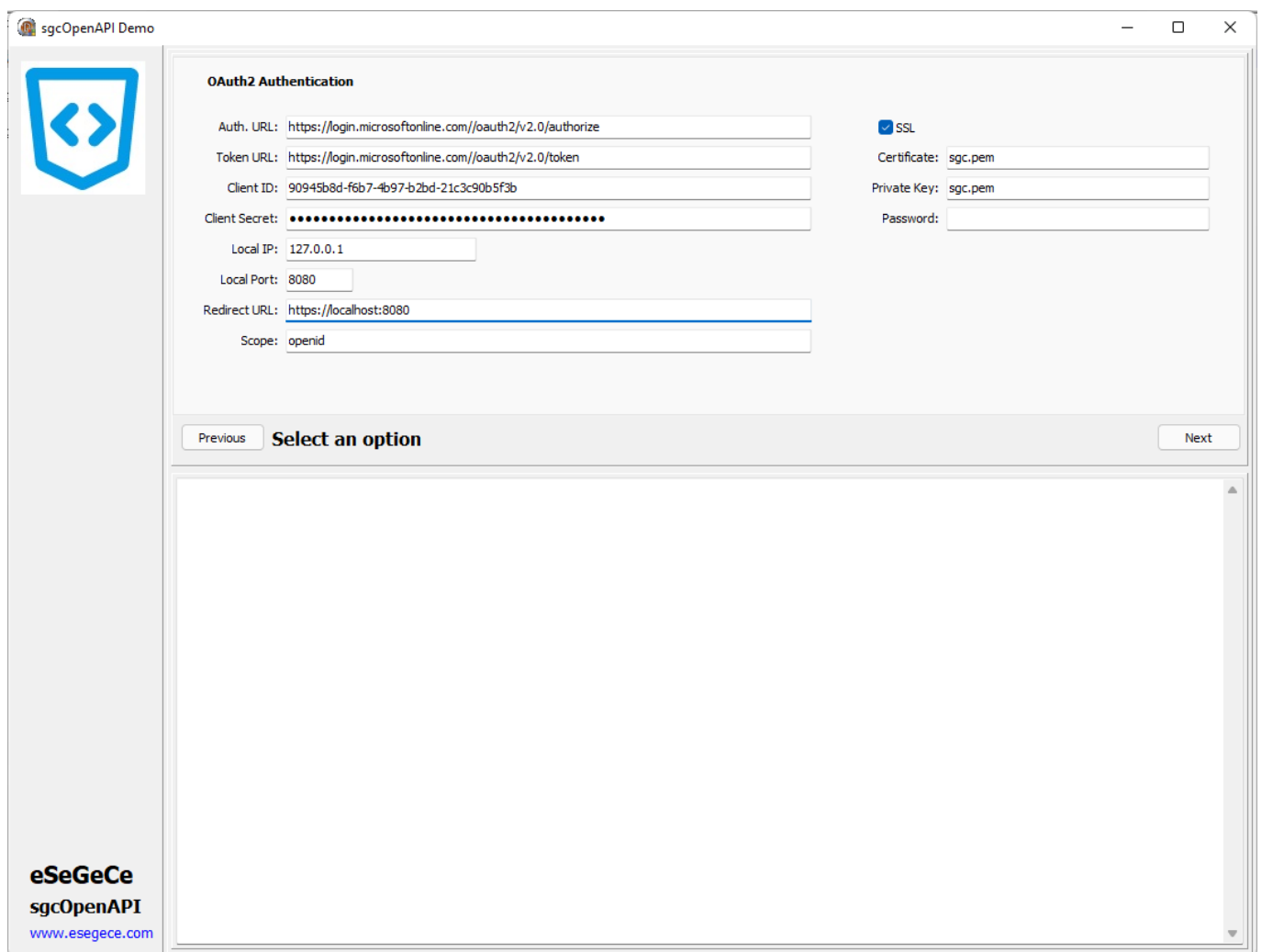
使用 OAuth2 授权码流程需要与用户交互，以便登录并获取所需权限。

获得 [租户 ID](#) 和 [凭据](#) 后，您可以为授权码授权方式配置 OAuth2 属性。

要为 OAuth2 代码授权配置 OpenAPI 客户端，请使用以下值配置属性 **MicrosoftOptions.Authentication**：

```
GetOpenAPIClient->MicrosoftOptions->Authentication = oamaOAuth2Code;
```

然后您可以配置 OAuth2 参数



sgcOpenAPI Demo

OAuth2 Authentication

Auth. URL:

Token URL:

Client ID:

Client Secret:

Local IP:

Local Port:

Redirect URL:

Scope:

☒ SSL

Certificate:

Private Key:

Password:

Select an option

eSeGeCe
sgcOpenAPI
www.esegece.com

```
GetOpenAPIClient->MicrosoftOptions->Authentication = oamaOAuth2Code;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->AuthURL = "https://login.microsoftonline.com/oauth2/v2.0/authorize";
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->TokenURL = "https://login.microsoftonline.com/oauth2/v2.0/token";
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  
";
```

必须为身份验证和令牌 URL 配置租户 ID，只需在 URL 中替换正确的租户 ID。

Microsoft 仅允许在本地 IP 监听时使用 **localhost** 作为 URL，因此请在**重定向 URL** 中使用 localhost 作为域名，而非配置为 IP 地址。

scope 值取决于 API，请查阅 Microsoft/Azure 文档了解每个 API 的信息。

第一次发起请求时，会显示网页浏览器，要求用户登录其 Microsoft 账户。如果用户之前已登录，则将直接发起 HTTP 请求。

OpenAPI Microsoft | OAuth2 凭据

使用 OAuth2 授权码授权流程无需与用户交互，因此适用于服务、守护进程或任何必须在无用户交互情况下运行的应用程序。

获得 [租户 ID](#) 和 [凭据](#)后，您可以为授权码授权方式配置 OAuth2 属性。

要为 OAuth2 代码授权配置 OpenAPI 客户端，请使用以下值配置属性 **MicrosoftOptions.Authentication**：

```
GetOpenAPIClient->MicrosoftOptions->Authentication = oamaOAuth2Credentials;
```

sgcOpenAPI Demo

OAuth2 Authentication

Auth. URL:

Token URL:

Client ID:

Client Secret:

Local IP:

Local Port:

Redirect URL:

Scope:

☒ SSL

Certificate:

Private Key:

Password:

Select an option

eSeGeCe
sgcOpenAPI
www.esegece.com

```
GetOpenAPIClient->MicrosoftOptions->Authentication = oamaOAuth2Credentials;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->AuthURL = "https://login.microsoftonline.com/oauth2/v2.0/authorize";
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->TokenURL = "https://login.microsoftonline.com/oauth2/v2.0/token";
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";
```

<https://graph.microsoft.com/.default>

```
";
```

必须为身份验证和令牌 URL 配置租户 ID，只需在 URL 中替换正确的租户 ID。

Microsoft 仅允许在本地 IP 监听时使用 **localhost** 作为 URL，因此请在**重定向 URL** 中使用 localhost 作为域名，而非配置为 IP 地址。

scope 值取决于 API，请查阅 Microsoft/Azure 文档了解每个 API 的信息。

OAuth2 凭证不需要任何用户交互，因此在第一次 HTTP 请求调用时不会打开浏览器。

OpenAPI Microsoft | Graph

Microsoft Graph 公开 REST API 和客户端库，用于访问以下 Microsoft 云服务中的数据：

- Microsoft 365 核心服务：Bookings、Calendar、Delve、Excel、Microsoft 365 合规 eDiscovery、Microsoft Search、OneDrive、OneNote、Outlook/Exchange、People（Outlook 联系人）、Planner、SharePoint、Teams、To Do、Workplace Analytics
- 企业移动性 + 安全服务：Advanced Threat Analytics、Advanced Threat Protection、Azure Active Directory、Identity Manager 和 Intune
- Windows 服务：活动、设备、通知、Universal Print
- Dynamics 365 Business Central

获取当前用户

```

GetOpenAPIClient->MicrosoftOptions->Authentication = oamaOAuth2Code;
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->AuthURL = "https://login.microsoftonline.cc
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->TokenURL = "https://login.microsoftonline.c
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientId = "90945b8d-f6b7-4b97-b2bd-21c3c90b5f3b";
GetOpenAPIClient->Authentication->OAuth2->OAuth2Options->ClientSecret = "client_secret";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->IP = "127.0.0.1";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->Port = 8080;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSL = True;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->Port = 8080;
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->CertFile = "sgc.pem";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->KeyFile = "sgc.pem";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->SSLOptions->Password = "";
GetOpenAPIClient->Authentication->OAuth2->LocalServerOptions->RedirectURL = "https://localhost:8080";
GetOpenAPIClient->Authentication->OAuth2->AuthorizationServerOptions->Scope->Text = "

openid

";
GetOpenAPIClient->meuserGetUser();
  
```

API

以下 API 是使用 eSeGeCe sgcOpenAPI 生成器生成的，并向所有用户免费提供。试用版附带接口的源代码。

API	描述
地理定位	Abstract IP 地理定位 API 接收 IP 地址，并将其转换为位置信息以及许多其他详细信

OpenAPI | AbstractApi 地理定位

什么是 IP 地理定位 API？

Abstract IP 地理定位 API 接受 IP 地址并将其转换为位置，以及许多其他详细信息，如地址、时区等。

IP 地理位置 API 有哪些使用场景？

IP 地理定位 API 及数据有许多强大的应用场景，包括但不限于：

- 根据用户位置自动将用户重定向到相关网站或子站点
- 自动检测并显示用户的位置、国家或时区，无需用户明确进行此定制。
- 根据用户位置自定义网站或应用程序的内容或体验。例如，显示用户当地的天气、税率和增值税率、货币、新闻、公众假期等。
- 根据用户位置过滤用户，例如，若您无法向特定国家/地区的用户提供服务。
- 要求用户接受特定条款，如欧盟公民所需的 GDPR Cookie 横幅等当地法规

在哪里获取 API Key？

只需在 [abstractapi.com](https://www.abstractapi.com) 注册，即可免费获取 API 密钥

<https://www.abstractapi.com/api/ip-geolocation-api>

示例代码

以下代码返回所提供 IP 地址的相关信息。

```
ShowMessage(GetOpenAPIClient->Retrieve_the_location_of_an_IP_address("asdfkjlkj32i3j2liwj3es", "88.5.12.4"));
```

演示 | 服务器聊天

本演示展示如何使用 [TsgcWebSocketHTTPServer](#) 和 [WebSocket](#) 作为通信协议构建服务器聊天室。每次有新的对等方发送消息时，服务器读取该消息并将其广播给所有已连接的客户端。

启动服务器

在启动服务器之前，您必须对其进行配置，设置监听端口，以及是否使用安全连接等.....

- 首先，我创建一个新的 [TsgcWebSocketHTTPServer](#) 实例。
- 若服务器将使用安全连接，则需要一个 PEM 证书；请设置该证书的位置以及 SSL 的监听端口。您可以根据需要配置 TLS 版本和 OpenSSL API。

```
// ... ssl
switch (cboOpenSSLAPI->ItemIndex)
{
    case 0:
        server->SSLOptions->OpenSSL_Options.APIVersion = TwsOpenSSLAPI->oslAPI_1_0;
        break;
    case 1:
        server->SSLOptions->OpenSSL_Options->APIVersion = TwsOpenSSLAPI->oslAPI_1_1;
        break;
}
switch (cboTLSVersion->ItemIndex)
{
    case 0:
        server->SSLOptions->Version = TwsTLSVersions->tlsUndefined;
        break;
    case 1:
        server->SSLOptions->Version = TwsTLSVersions->tls1_0;
        break;
    case 2:
        server->SSLOptions->Version = TwsTLSVersions->tls1_1;
        break;
    case 3:
        server->SSLOptions->Version = TwsTLSVersions->tls1_2;
        break;
    case 4:
        server->SSLOptions->Version = TwsTLSVersions->tls1_3;
        break;
    default:
        break;
}
```

- 默认情况下，启动服务器后，它将监听所有 IP 的监听端口，因此建议使用 binding 属性仅监听 1 个特定 IP。

```
With WSServer->Bindings->Add do
{
    Port = StrToInt(txtDefaultPort->Text);
    IP = txtHost->Text;
}
```

- 配置完所有选项后，调用 `Server.Active = true` 以启动服务器。

事件配置

- 使用 **OnConnect** 和 **OnDisconnect** 事件可以了解客户端何时连接到服务器。
- 消息从 **客户端发送到服务器** 通过 **OnMessage** 事件接收，因此使用此事件处理程序将收到的消息广播给所有客户端


```
private void OnMessageEvent(TsgcWSConnection *Connection, string Text)
{
    server->Broadcast(Text);
}
```

分发 HTTP 请求

WebSocket HTTP 服务器允许您在同一监听端口上处理 WebSocket 和 HTTP 协议，因此 Web 浏览器可以请求网页来访问您的服务器。**OnCommandGet** 是用于读取 HTTP 请求并发送 HTTP 响应的事件。

使用 `ARequestInfo` 参数读取 HTTP 请求，使用 `AResponseInfo` 写入 HTTP 响应。

基本上，使用 `ARequestInfo.Document` 读取客户端请求的文档，并使用以下属性发送响应：`ResponseNo`、`ContentType` 和 `ContentText`。

示例：客户端请求文档 '/jquery.js'

```
private void OnCommandGetEvent(TsgcWSConnection *Connection, TsgcWSHTTPRequestInfo *RequestInfo, ref TsgcWSHTTPRe
{
    if (RequestInfo->Document == "/jquery.js")
    {
        ResponseInfo->ContentType = "text/javascript";
        ResponseInfo->ContentText = pagejQuery.Content;
        ResponseInfo->ResponseNo = 200;
    }
}
```

客户端聊天

此演示展示了如何使用 [TsgcWebSocketClient](#) 构建一个客户端聊天，该客户端连接到 WebSocket 服务器，发送消息，所有已连接的客户端均可接收到该消息。

连接到服务器

- 首先创建 [TsgcWebSocketClient](#) 的新实例。
- 然后配置服务器的 **Host** 和 **Port**。
- 如果客户端使用安全连接，请配置 组件的 **TLSOptions** 属性。

```
if (chkTLS->Checked)
{
    WSClient->Port = StrToInt(txtSSLPort->Text)
}
else
{
    WSClient->Port = StrToInt(txtDefaultPort->Text);
}
WSClient->Host = txtHost->Text;
case cboOpenSSLAPI->ItemIndex of
{
    0: WSClient->TLSOptions->OpenSSL_Options->APIVersion = sslAPI_1_0;
    1: WSClient->TLSOptions->OpenSSL_Options->APIVersion = sslAPI_1_1;
};
case cboTLSVersion->ItemIndex of
{
    0: WSClient->TLSOptions->Version = tlsUndefined;
    1: WSClient->TLSOptions->Version = tls1_0;
    2: WSClient->TLSOptions->Version = tls1_1;
    3: WSClient->TLSOptions->Version = tls1_2;
    4: WSClient->TLSOptions->Version = tls1_3;
}
WSClient->TLS = chkTLS->Checked;
```

- 所有选项配置完成后，将 **Client.Active** 设为 **true** 即可连接到服务器。

向服务器发送消息

- 要向服务器发送消息，请使用 **WriteData** 方法，发送任意文本消息后，服务器将以相同消息作为响应回复。

```
WSClient->WriteData(txtName->Text + ": " + txtMessage->Text);
```

从服务器接收消息

- 每当客户端收到新的文本消息时，**OnMessage** 事件被触发。

```
private void OnMessageEvent(TsgcWSConnection *Connection, string Text)
{
    DoLog(Text);
}
```

Demos | 客户端

此演示展示了如何使用 [TsgcWebSocketClient](#) 构建 WebSocket 客户端。

连接到服务器

- 首先创建 [TsgcWebSocketClient](#) 的新实例。
- 然后配置服务器的 **Host** 和 **Port**。
- 默认情况下，客户端将使用 **WebSocket** 协议进行连接。但您也可以将客户端配置为使用**纯 TCP** 协议连接。只需将 **Specifications.RFC6455 = false**，客户端将改用纯 TCP 协议而非 WebSocket 协议。您可以阅读关于 [TCP 连接](#) 的详细信息。

```
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;
```

客户端事件

使用以下事件控制客户端流程：连接时、断开时、接收消息时、检测到错误时.....

```
private void OnExceptionEvent(TsgcWSConnection *Connection, Exception E)
{
    DoLog("#exception: " + E->Message);
}
private void OnConnectEvent(TsgcWSConnection *Connection)
{
    DoLog("#connected: " + Connection->IP);
}
private void OnMessageEvent(TsgcWSConnection *Connection, string Text)
{
    DoLog(Text);
}
private void OnDisconnectEvent(TsgcWSConnection *Connection, int CloseCode)
{
    DoLog("Disconnected (" + IntToStr(CloseCode) + "): " + Connection->IP);
}
private void OnErrorEvent(TsgcWSConnection *Connection, string Error)
{
    DoLog("#error: " + Connection->IP + " - " + Error);
}
```

演示 | 客户端 MQTT

此演示展示了如何连接到 MQTT 代理服务器。需要 [TsgcWebSocketClient](#) 来处理 WebSocket/TCP 协议。

配置

- 首先创建一个新的 [TsgcWebSocketClient](#) 实例，查看[客户端演示](#)。
- 然后，创建一个新的 [TsgcWSPClient_MQTT](#) 实例。
- 之后，您必须将 MQTT 协议 分配给 WebSocket 客户端，并在 WebSocket 客户端中配置连接选项。

```
if (mqtt == null)
{
    mqtt = new TsgcWSPClient_MQTT();
    mqtt->OnMQTTBeforeConnect += OnMQTTBeforeConnectEvent;
    mqtt->OnMQTTConnect += OnMQTTConnectEvent;
    mqtt->OnMQTTDisconnect += OnMQTTDisconnectEvent;
    mqtt->OnMQTTSubscribe += OnMQTTSubscribeEvent;
    mqtt->OnMQTTUnSubscribe += OnMQTTUnSubscribeEvent;
    mqtt->OnMQTTPing += OnMQTTPingEvent;
    mqtt->OnMQTTPubAck += OnMQTTPubAckEvent;
    mqtt->OnMQTTPubComp += OnMQTTPubCompEvent;
    mqtt->OnMQTTPublish += OnMQTTPublishEvent;
    mqtt->OnMQTTPubRec += OnMQTTPubRecEvent;
    mqtt->Client = client;
}
mqtt->Client = client;
txtParameters->Text = "/";
chkTLS->Checked = false;
mqtt->Authentication->Enabled = false;
mqtt->Authentication->UserName = "";
mqtt->Authentication->Password = "";
mqtt->MQTTVersion = TwsmqttVersion->mqtt311;
mqtt->HeartBeat->Interval = 5;
mqtt->HeartBeat->Enabled = true;
switch (Index)
{
    case 0: // esegece.com
        txtHost->Text = "www.esegece.com";
        txtPort->Text = "15675";
        txtParameters->Text = "/ws";
        mqtt->Authentication->Enabled = true;
        mqtt->Authentication->UserName = "sgc";
        mqtt->Authentication->Password = "sgc";
        chkOverWebSocket->Checked = true;
        break;
    case 1: // test.mosquitto.org
        txtHost->Text = "test.mosquitto.org";
        txtPort->Text = "1883";
        chkTLS->Checked = false;
        chkOverWebSocket->Checked = false;
        break;
    case 2: // mqtt.fluux.io
        txtHost->Text = "mqtt.fluux.io";
        txtPort->Text = "1883";
        chkTLS->Checked = false;
        chkOverWebSocket->Checked = false;
        mqtt->MQTTVersion = TwsmqttVersion->mqtt5;
        break;
    case 3: // broker.hivemq.com
        txtHost->Text = "broker.mqttdashboard.com";
        txtPort->Text = "8000";
        txtParameters->Text = "/mqtt";
        chkTLS->Checked = false;
        chkOverWebSocket->Checked = true;
        mqtt->MQTTVersion = TwsmqttVersion->mqtt5;
        break;
}
```

MQTT 事件

连接流程由 MQTT 客户端组件控制，因此您必须处理 MQTT 事件，以了解何时连接到代理、何时发布新消息、何时断开连接...

```
private void OnMQTTConnectEvent(TsgcWSConnection *Connection, bool Session, int ReasonCode, string ReasonName, Ts
{
    DoLog("#MQTT Connect");
    chkTLS->Enabled = false;
    chkCompressed->Enabled = false;
    if (FMQTTSubscribeTopic != "")
    {
        mqtt->Subscribe(FMQTTSubscribeTopic);
        FMQTTSubscribeTopic = "";
    }
}

private void OnMQTTPublishEvent(TsgcWSConnection *Connection, string Topic, string Text, TsgcWSMQTTPublishPropert
{
    DoLog(Topic + ": " + Text);
}

private void OnMQTTSubscribeEvent(TsgcWSConnection *Connection, int PacketIdentifier, TsgcWSSUBACKS *Codes, TsgcW
{
    DoLog("#Subscribe: " + IntToStr(PacketIdentifier));
}

private void OnMQTTDisconnectEvent(TsgcWSConnection *Connection, int ReasonCode, string ReasonName, TsgcWSMQTTDIS
{
    DoLog("#disconnected");
    chkTLS->Enabled = true;
    chkCompressed->Enabled = true;
}
```

演示 | 客户端 SocketIO

此演示展示了如何连接到 Socket.IO 服务器。需要一个 [TsgcWebSocketClient](#) 来处理 WebSocket/TCP 协议。

配置

- 首先创建一个新的 [TsgcWebSocketClient](#) 实例，查看[客户端演示](#)。
- 然后，创建 [TsgcWSAPI_SocketIO](#) 的新实例。
- 之后，您必须将 Socket.IO API 分配给 WebSocket 客户端，并在 WebSocket 客户端中配置连接选项。

```
if (socketio == null)
{
    socketio = new TsgcWSAPI_SocketIO();
    socketio->Client = client;
}
socketio->Client = client;
txtParameters->Text = "/";
chkTLS->Checked = true;
chkOverWebSocket->Checked = true;
```

发送消息

Socket.IO 使用 **TsgcWebSocketClient** 向服务器发送消息，只需调用 **WriteData** 并将 JSON 消息作为参数传递给 socket.io 服务器

```
client->WriteData("42[\"new message\", \"\" + txtSocketIOMessage.Text + "\"]");
```

接收消息

作为连接流程的消息由 TsgcWebSocketClient 处理，因此请使用此组件读取服务器发送的消息，并了解连接是否处于活跃状态。

```
private void OnMessageEvent(TsgcWSConnection *Connection, string Text)
{
    DoLog(Text);
}
```

Demos | 服务器监控

此演示展示了如何使用 WebSocket 协议更新 3 个 HTML 监视器。服务器有一个内部计时器，随机更新仪表盘的值，并使用 WebSocket 消息更新该值。该消息由 JavaScript 客户端读取并更新仪表盘的值。

配置

- 首先创建一个新的 [TsgcWebSocketServer](#) 实例，参考[服务器聊天演示](#)。
- 然后创建一个新计时器，每 500 毫秒更新一次内存、网络或 CPU 的值，并将更新发送给所有已连接的客户端。
- 在 JavaScript 客户端中，读取 服务器发送的消息并更新仪表盘的值。

```
<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 sgws("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('sgcvent', 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-left">
    </div><!-- /header -->
    <div data-role="content">
        <h2>Press Start to Get Monitor Data</h2>
        <p id="status" class="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" onclick=
                <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>

```


示例 | 服务器快照

本演示展示如何从服务器向客户端发送图像，以及如何利用服务器组件的广播方法使所有客户端接收同一张图像。

配置

- 首先创建一个新的 [TsgcWebSocketHTTPServer](#) 实例，查看 [服务器聊天演示](#)。
- 启用压缩，以减少向客户端传输消息时的字节数

```
Server.Extensions.PerMessage_Deflate.Enabled = true
```

- 之后，服务器每 5 秒向所有已连接的客户端广播一个图像流

```
void DoBroadcastStream()
{
    TBitmap *oBitmap = new TBitmap();
    try
    {
        // load bitmap
        if (WSServer->Active)
        {
            TMemoryStream *oStream = new TMemoryStream();
            Try
            {
                oBitmap->SaveToStream(oStream);
                oStream->Seek(0, soFromBeginning);
                WSServer->Broadcast(oStream);
            }
            __finally
            {
                FreeAndNil(oStream);
            }
        }
    }
    __finally
    {
        oBitmap->FreeImage;
        FreeAndNil(oBitmap);
    }
}
```

演示 | 客户端快照

此演示展示了如何读取二进制 WebSocket 消息，使用 [TsgcWebSocketClient](#) 连接到 WebSocket 服务器，并接收作为图像的流，最终显示给用户。

连接到服务器

- 首先创建 [TsgcWebSocketClient](#) 的新实例。
- 然后配置服务器的 **Host** 和 **Port**。
- 启用压缩，以在消息从服务器传输时减少字节数。

```
Client.Extensions.PerMessage_Deflate.Enabled = true
```

- 服务器发送的图像以流的形式到达，因此使用 **OnBinary** 事件读取图像。

```
void OnClientBinary(TsgcWSConnection *Connection, TMemoryStream *Data)
{
    TBitmap *oBitmap = new TBitmap();
    try
    {
        oBitmap->LoadFromStream(Data);
        Image1->Picture->Assign(oBitmap);
        memoLog->Lines->Add(
            "#image uncompressed size: " + IntToStr(Data->Size) +
            ". Total received: " + IntToStr(Connection->RecBytes));
    }
    __finally
    {
        FreeAndNil(oBitmap);
    }
}
```

示例 | 上传文件

此演示展示了如何使用 WebSocket 协议从浏览器将文件上传到服务器。

配置

- 首先创建一个新的 [TsgcWebSocketServer](#) 实例，参考[服务器聊天演示](#)。
- 文件将以二进制流的形式到达服务器，因此您必须处理 **OnBinary** 事件来读取文件。

```
void OnServerBinary(TsgcWSConnection *Connection, const TMemoryStream *Data)
{
    if (FFileName == "")
        FFileName = FormatDateTime("yyyymmddhhnnsszz", Now) + ".dat";
    TFileStream *oFile = new TFileStream(FFileName, fmCreate);
    try
    {
        oFile->CopyFrom(Data, Data->Size);
    }
    finally
    {
        delete oFile;
    }
    memoLog->Lines->Add("Received File: " + FFileName);
}
```

- 如果您想知道文件名，可以在文件发送前发送一条包含文件名的文本消息

```
void OnServerMessage(TsgcWSConnection *Connection, const String Text)
{
    if (LeftStr(Text, Length(CS_UPLOAD_FILE)) == CS_UPLOAD_FILE)
        FFileName = MidStr(Text, Length(CS_UPLOAD_FILE) + 1, Length(Text));
    else
        memoLog->Lines->Add("Message Received (" + Connection->Guid + "): " + Text);
}
```

用于通过 **WebSockets** 发送文件的 **JavaScript** 代码如下所示：

```
<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.onloadend = 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>
```

演示 | 服务器身份验证

此演示展示如何使用服务器身份验证。如需了解不同类型的身份验证，请阅读以下关于 [Authentication](#) 的文章。

身份验证

- 首先创建 `TsgcWebSocketServer` 的新实例。启用 `Authentication` 属性：`server.Authentication.Enabled = true`；
- 然后，在 `OnAuthentication` 事件处理程序中检查用户名和密码是否正确。如果正确，将 `Authenticated` 属性设置为 `true`，否则设置为 `false`。

```
private void OnAuthenticationEvent(TsgcWSConnection *Connection, string User, string Password, ref bool Authenticated)
{
    if ((User == "user") && (Password == "1234"))
    {
        Authenticated = true;
    }
}
```

演示 | KendoUI_Grid

该演示展示了 KendoUI Grid 如何使用 WebSocket 作为协议、以 Web 浏览器作为客户端工作。基本上这是一个 JavaScript 网格，当任意客户端进行更改时即时更新，这些更改通过 WebSocket 协议推送给所有已连接的客户端，因此所有客户端都能实时看到相同的数据，包括客户端所做的所有更改。

配置

- 首先创建一个新的 [TsgcWebSocketHTTPServer](#) 实例，查看 [服务器聊天演示](#)。
- 然后，您必须处理 OnCommandGet 事件，以发送 Web 浏览器客户端请求的所需文件。

```
void OnCommandGet(TidContext *AContext, TidHttpRequestInfo *ARequestInfo, TidHttpResponseInfo *AResponseInfo)
{
    if (ARequestInfo->Document == "/jquery.mobile.css")
    {
        AResponseInfo->ContentText = pagejQueryMobileCSS->Content;
        AResponseInfo->ContentType = "text/css";
        AResponseInfo->ResponseNo = 200;
    }
    else if (ARequestInfo->Document == "/jquery.js")
    {
        AResponseInfo->ContentText = pagejQuery->Content;
        AResponseInfo->ContentType = "text/javascript";
        AResponseInfo->ResponseNo = 200;
    }
    else if (ARequestInfo->Document == "/jquery.mobile.js")
    {
        AResponseInfo->ContentText = pagejQueryMobile->Content;
        AResponseInfo->ContentType = "text/javascript";
        AResponseInfo->ResponseNo = 200;
    }
    else
    {
        if (AContext->Connection->Socket->Binding->Port == WSServer->SSLOptions->Port)
            FRequestSSL = true;
        else
            FRequestSSL = false;
        AResponseInfo->ContentText = pageKendoUI_Grid->Content;
        AResponseInfo->ContentType = "text/html";
        AResponseInfo->ResponseNo = 200;
    }
}
```

WebSockets 更新

当客户端更新网格记录时，此变更将通过 WebSocket 协议传输给所有已连接的客户端。使用 OnMessage 事件可获取网格变更通知。消息为 JSON 格式，您只需读取 JSON 文本、解码并向对端发送响应。

```
void OnServerMessage(TsgcWSConnection *Connection, const String Text)
{
    TsgcJSON *oJSON = new TsgcJSON();
    try
    {
        oJSON->Read(Text);
        // ... read
        if (oJSON->Node["type"]->Value == "read")
        {
            TsgcObjectJSON *oArray = oJSON->AddArray("data");
            for (int i = 0; i < 20; i++)
            {
                TsgcJSON *oObj = oArray->JSONObject->AddObject(IntToStr(i))->JSONObject;
                oObj->AddPair("ContactID", i);
                oObj->AddPair("ContactName", ContactName[i]);
                oObj->AddPair("ContactTitle", ContactTitle[i]);
                oObj->AddPair("CompanyName", CompanyName[i]);
                oObj->AddPair("Country", Country[i]);
            }
        }
    }
}
```

```

    }
    Connection->WriteData(oJSON->Text);
}
// ... update
else if (oJSON->Node["type"]->Value == "update")
{
    WSServer->Broadcast(StringReplace(Text, "\"type\": \"update\"", "\"type\": \"push-update\"", []), "", "",
        Connection->Guid);
    Connection->WriteData(Text);
}
// ... destroy
else if (oJSON->Node["type"]->Value == "destroy")
{
    WSServer->Broadcast(StringReplace(Text, "\"type\": \"destroy\"", "\"type\": \"push-destroy\"", []), "", "",
        Connection->Guid);
    Connection->WriteData(Text);
}
// ... create
else if (oJSON->Node["type"]->Value == "create")
{
    String vText = StringReplace(Text, "null", FormatDateTime("yyyymmddhhnnsszzz", Now), []);
    WSServer->Broadcast(StringReplace(vText, "\"type\": \"create\"", "\"type\": \"push-create\"", []), "", "",
        Connection->Guid);
    Connection->WriteData(vText);
}
}
finally
{
    oJSON->Free();
}
}

```

演示 | ServerSentEvents

此演示展示了服务器发送事件在 WebSocket 服务器中的工作原理。sgcWebSockets 允许服务器在同一监听端口上处理多个协议。

您可以进一步了解[服务器发送事件](#)。

此演示展示服务器如何使用 Server Sent Events 每秒向所有已连接客户端发送当前时间。

服务器启动后，向所有已连接客户端广播一条包含服务器时间的消息，因此每次客户端收到此消息时都会显示给用户。

```
private void Timer1Timer(TObject *sender)
{
    server->Broadcast("data: " + "Server Time: " + FormatDateTime("hh:nn:ss", Now));
}
```

处理 WebSocket 连接的 JavaScript 代码如下所示：

```
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";
});
```


演示 | 服务器 WebRTC

此演示展示了如何使用 [TsgcWebsocketHTTPServer](#) 和作为 JavaScript 库的 WebRTC 构建视频会议服务器。

演示使用 WebSocket 协议对 WebRTC 进行信令，并使用公共 STUN/TURN 服务器。对于生产站点，您需要使用自己的 STUN/TURN 服务器。注册用户可以下载适用于 Windows 的 Coturn，该版本已编译完成，包含在您的服务器上运行所需的所有库。

客户端必须是支持 WebRTC 连接的 Web 浏览器。

配置

- 首先创建一个新的 [TsgcWebsocketHTTPServer](#) 实例，查看 [服务器聊天演示](#)。
- 然后，创建一个新的 [TsgcWSPServer_WebRTC](#) 实例。
- 此后，您必须将 WebRTC 协议分配给 WebSocket 服务器，并配置服务器的主机和端口。

```
WSServer->Port = StrToInt(txtDefaultPort->Text);  
// ... bindings  
With WSServer->Bindings->Add do  
{  
    Port = StrToInt(txtDefaultPort->Text);  
    IP = txtHost>Text;  
}  
// ... active  
WSServer->Active = true;
```

该演示需要一个索引 HTML 页面，用于分发 WebRTC 前端页面，该页面随演示一起提供。

在 WebBrowser 中运行

配置好服务器后，启动服务器并选择一款可用的 Web 浏览器。浏览器将打开一个新会话，要求启动新会话。成功后您将看到自己的视频；如果在另一个 Web 浏览器中打开相同的 URL，则可以看到两个对等端均已连接。

示例默认不使用 SSL 运行，这仅适用于 localhost 连接。对于生产站点，请使用 SSL 连接。参见 [服务器聊天示例](#)，了解如何在服务器端配置 SSL。

演示 | 服务器 AppRTC

此演示展示如何使用 [TsgcWebSocketHTTPServer](#) 和 AppRTC JavaScript 库构建视频会议服务器。

演示使用 WebSocket 协议对 WebRTC 进行信令，并使用公共 STUN/TURN 服务器。对于生产站点，您需要使用自己的 STUN/TURN 服务器。注册用户可以下载适用于 Windows 的 Coturn，该版本已编译完成，包含在您的服务器上运行所需的所有库。

客户端必须是支持 WebRTC 连接的 Web 浏览器。

配置

- 首先创建一个新的 [TsgcWebSocketHTTPServer](#) 实例，查看 [服务器聊天演示](#)。
- 然后，创建一个新的 [TsgcWSPServer_AppRTC](#) 实例。
- 之后，您必须将 AppRTC 协议分配给 WebSocket 服务器并配置服务器主机和端口。WebRTC 需要安全连接，因此您需要使用 PEM 证书并配置组件的 SSLOptions 属性。

```
WSServer->Port = StrToInt(txtDefaultPort->Text);
// ... bindings
With WSServer->Bindings->Add do
{
    Port = StrToInt(txtDefaultPort->Text);
    IP = txtHost>Text;
}
// ... properties
WSPAppRTC->AppRTC->RoomLink = "https://" + txtHost->Text + ":" + txtDefaultPort->Text + "/r/";
WSPAppRTC->AppRTC->WebSocketURL = "wss://" + txtHost->Text + ":" + txtDefaultPort->Text;
// ... active
WSServer->Active = true;
```

- AppRTC.RoomLink 是 Web 浏览器将被重定向到以登录房间的 URL
- AppRTC.WebSocketURL 是 WebSocket 连接的 URL
- IceServers 可在 AppRTC 服务器协议中进行配置。

该演示需要一个用于调度 AppRTC 前端页面的索引 HTML 页面，该页面随演示一起提供。

在 WebBrowser 中运行

服务器配置完成后，启动服务器并选择一个可用的 Web 浏览器。系统将打开一个新的浏览器会话，提示加入新房间。加入后若成功，您将看到一个链接，需在另一个浏览器中使用该链接以发起新的视频会议。

AppRTC

Please enter a room name.

959454873

JOIN

RANDOM

Recently used rooms:

演示 | Telegram 客户端

此演示展示如何连接到 Telegram、接收所有联系人、发送文本消息、发送图片等更多功能。

配置

- 首先创建 `TsgcTDLib_Telegram` 的新实例。
- 然后，在尝试连接 Telegram 之前，您必须向客户端组件传递一些参数，如 API Hash、API Id 等。设置好所有必填参数后，将属性 `Active` 设置为 `true` 以启动连接。

```
telegram->Telegram->API->ApiHash = txtApiHash->Text;
telegram->Telegram->API->ApiId = txtApiId->Text;
telegram->Telegram->PhoneNumber = "";
telegram->Telegram->BotToken = "";
if (chkLoginBot->Checked)
{
    telegram->Telegram->BotToken = txtBotToken->Text;
}
else
{
    telegram->Telegram->PhoneNumber = txtPhoneNumber->Text;
}

telegram->Active = true;
```

- 当客户端尝试连接到 Telegram 时，通常需要验证码，因此您必须处理 `OnTelegramAuthenticationCode` 事件，并在 `Code` 参数中返回您 Telegram 账户提供的值。

```
private void OnAuthenticationCodeEvent(TObject *Sender, ref string Code)
{
    Code = InputBox("Telegram", "Introduce Telegram Code");
}
```

发送 Telegram 消息

发送 Telegram 消息（文本、文件、图片...）时，始终需要先设置要发送消息的 `ChatId`，然后设置参数（可以是文本消息、文件名...）。

```
// send text message
sgcTelegram->SendTextMessage("456413", "Hello From sgcWebSockets!!!");

// send file message
sgcTelegram->SendDocumentMessage("383784", "c:\\yourfile.txt");
```

接收 Telegram 消息

Telegram 客户端接收到的消息由特定的事件处理程序处理。有一个事件用于接收下一条文本消息，另一个用于接收新文档、照片...

```
private void OnMessageTextEvent(TObject *Sender, TsgcTelegramMessageText *MessageText)
{
    DoLogMessage(MessageText->ChatId, MessageText->SenderUserId->ToString(), MessageText->Text);
}
```

```
private void OnMessageDocumentEvent(TsgcTDLib_Telegram *Sender, TsgcTelegramMessageDocument *MessageDocument)
{
    DoLogMessage(MessageDocument->ChatId, MessageDocument->SenderUserId->ToString(), MessageDocument->FileName);
}
```

Coturn

Coturn

从 **sgcWebSockets 4.5.2 企业版**起，您可以使用 **Delphi/CBuilder** 构建自己的 **STUN/TURN 服务器**。

这是一个免费的 TURN 和 STUN 服务器开源实现。

TURN 服务器是一个 VoIP 媒体流量 NAT 穿透服务器和网关。它也可以用作通用网络流量 TURN 服务器和网关。

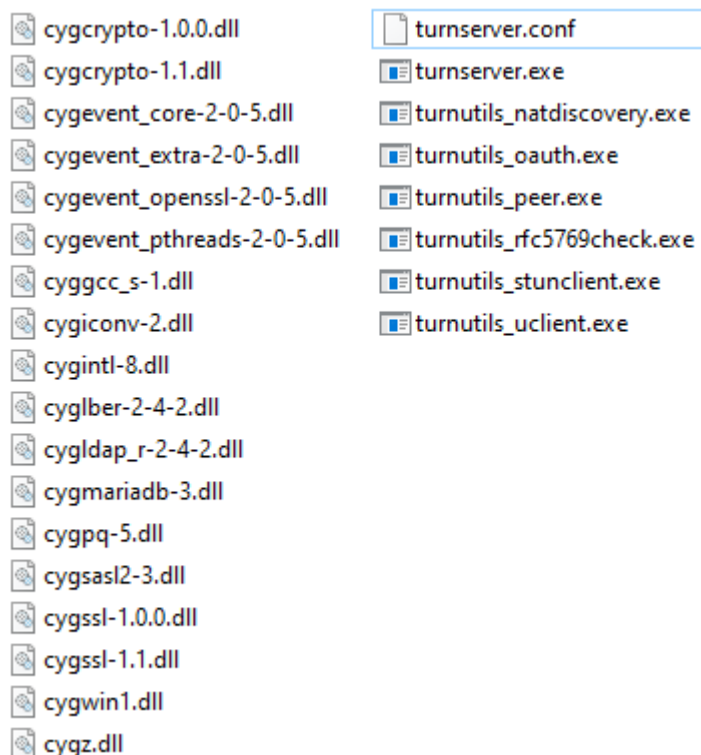
支持的项目目标平台为：

- **Linux**
- **Mac OS X**
- **Windows**（Cygwin）：已编译的二进制文件面向注册用户提供。

Windows 配置

首先，您必须从账户下载编译好的二进制文件，有两个可用版本：win32 和 win64。选择所需平台并在文件夹中解压二进制文件。将创建以下文件：

1. 运行应用程序所需的一些 cygwin 库，您必须随 coturn 服务器一起部署这些库。
2. 某些控制台应用程序：
 - 2.1 turnserver.exe：这是运行 TURN/STUN 服务器的主控制台应用程序。
 - 2.2 其他应用程序：用于配置 或测试目的。
3. Turnserver.conf：是 coturn 服务器的配置文件。



turnserver.conf

这是 coturn 服务器的配置文件，打开后您将看到默认配置。

简单配置

您的服务器公网 IP 为 80.15.44.123，侦听端口 80。连接凭证为：用户名 = demo，密码 = secret。
设置以下配置：

```
listening-ip=80.15.44.123
listening-port=80
realm=yourrealm.com
user=demo:secret
```

启用 TLS 的配置

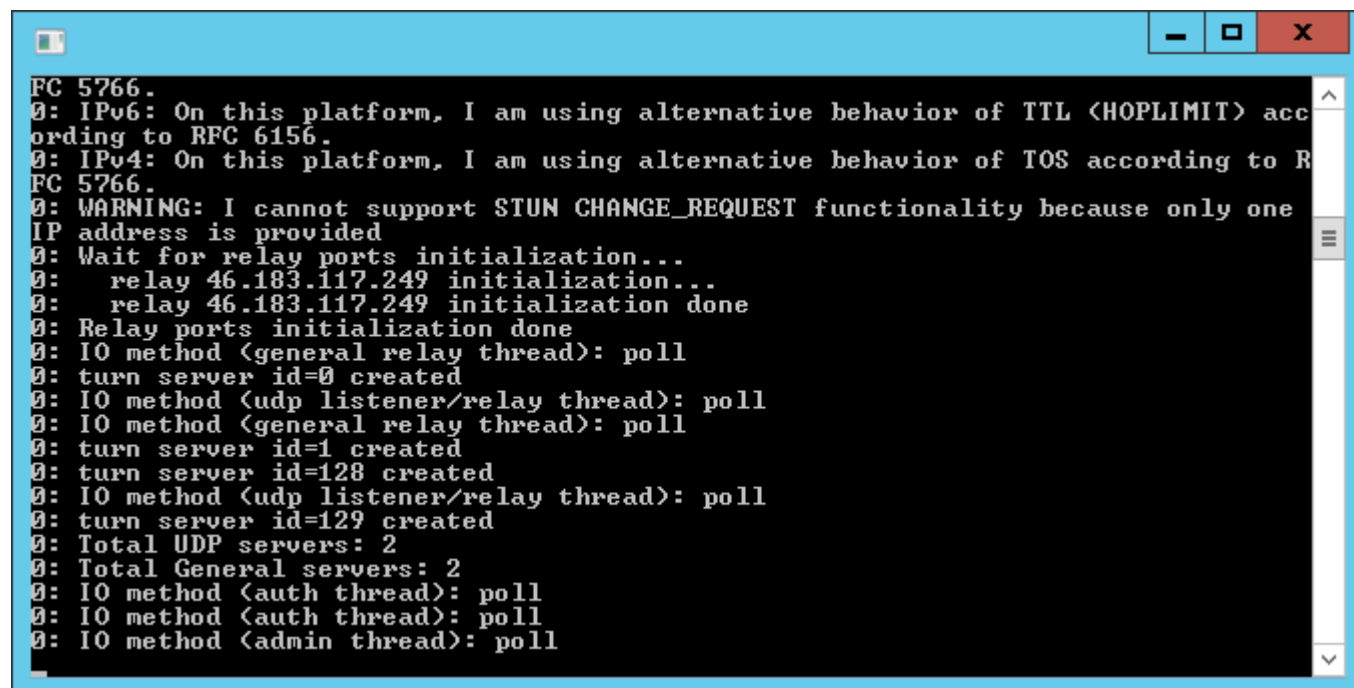
服务器公网 IP 为 80.15.44.123，监听 80 端口和 443 端口（TLS 连接）。连接凭据：用户名 = demo，密码 = secret。您的证书名称（必须为 PEM 格式）为 certificate.crt，私钥为 private.key。
设置以下配置：

```
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
```

还有更多可用配置，只需打开 turnserver.conf 并阅读有文档说明的部分。

运行 coturn

配置完成后，直接运行 turnserver.exe 即可启动服务器，此时将打开一个新的控制台应用程序并创建日志文件。如果在 turnserver.conf 文件中启用"verbose"，可以增加控制台应用程序的详细程度（获取更详细的信息）。



```
FC 5766.
0: IPv6: On this platform, I am using alternative behavior of TTL <HOPLIMIT> according to RFC 6156.
0: IPv4: On this platform, I am using alternative behavior of TOS according to RFC 5766.
0: WARNING: I cannot support STUN CHANGE_REQUEST functionality because only one IP address is provided
0: Wait for relay ports initialization...
0:   relay 46.183.117.249 initialization...
0:   relay 46.183.117.249 initialization done
0: Relay ports initialization done
0: IO method <general relay thread>: poll
0: turn server id=0 created
0: IO method <udp listener/relay thread>: poll
0: IO method <general relay thread>: poll
0: turn server id=1 created
0: turn server id=128 created
0: IO method <udp listener/relay thread>: poll
0: turn server id=129 created
0: Total UDP servers: 2
0: Total General servers: 2
0: IO method <auth thread>: poll
0: IO method <auth thread>: poll
0: IO method <admin thread>: poll
```

许可证

eSeGeCe 组件最终用户许可协议

eSeGeCe 组件最终用户许可协议（EULA）是您与 eSeGeCe 作者之间关于所有 eSeGeCe 组件的法律协议。安装、复制或以其他方式使用 eSeGeCe 组件，即表示您同意受本 EULA 条款的约束。如果您不同意本 EULA 的条款，请勿安装或使用 eSeGeCe 组件。

eSeGeCe 组件受版权法和国际版权条约以及其他知识产权法律和条约的保护。eSeGeCe 组件属于许可使用，而非出售。

如果您需要源代码，则需要支付注册费。您不得将许可证密钥和/或 eSeGeCe 完整版本（包括 DCU 版本和源代码版本）提供给任何第三方个人和/或实体。您也不得使用来自任何第三方个人和/或实体的许可证密钥和/或 eSeGeCe 完整版本。

1. 许可证授予

eSeGeCe 组件的许可方式如下：

(a) 安装与使用。

许可方授予您在运行有效许可的操作系统副本的计算机上安装和使用 eSeGeCe 组件的权利，该操作系统须为 eSeGeCe 组件所设计的系统 [例如 Windows 2000、Windows 2003、Windows XP、Windows ME、Windows Vista、Windows 7、Windows 8、Windows 10]。

(b) 免版税。

您可以基于 eSeGeCe 组件创建商业应用程序，并将其与您的可执行文件一起分发，无需支付版税。

(c) 修改（仅限源码版本）。

您可以在本许可协议规定的条款下，对提供给您的授权源代码进行修改、增强、衍生作品及/或扩展。

(d) 备份副本。

您也可以根据需要复制 eSeGeCe 组件，用于备份和归档目的。

2. 其他权利和限制的说明

(a) 版权声明的维护。

您不得删除或修改 eSeGeCe 组件任何副本上的任何版权声明。

(b) 分发。

您不得将 eSeGeCe 组件的注册副本分发给第三方。可从 eSeGeCe 官方网站下载的评估版可以自由分发。

您可以为您的应用程序创建包含 eSeGeCe 组件的组件/ActiveX 控件/库，但不得将其分发或发布给第三方。

(c) 禁止分发源代码（仅限源码版）。

您不得以源代码形式向第三方分发或发布源代码，或其任何修改、增强、衍生作品和/或扩展。

您不得以任何方式分发、发布、披露或以其他方式向第三方提供源代码的任何部分。

(d) 禁止反向工程、反编译和反汇编。

您不得对 eSeGeCe 组件进行逆向工程、反编译或反汇编，仅限适用法律明确允许的情形，且该限制不受本条款约束。

(e) 租用。

您不得出租、租赁或借出 eSeGeCe 组件。

(f) 支持服务。

许可方可能为您提供与 eSeGeCe 组件相关的支持服务（“支持服务”）。作为支持服务的一部分提供给您的任何补充软件代码均视为 eSeGeCe 组件的一部分，并受本最终用户许可协议条款和条件的约束。

eSeGeCe 一次只允许一位开发人员使用，技术支持也仅提供给特定的一位开发人员。

(g) 遵守适用法律。

您必须遵守所有适用法律，规范 eSeGeCe 组件的使用。

3. 终止

在不影响其他权利的情况下，如果您违反本 EULA 的条款和条件，LICENSER 可终止本 EULA。在此情况下，您必须销毁您所持有的 eSeGeCe 组件的所有副本。

4. 版权

eSeGeCe 组件及其任何副本的所有权，包括但不限于版权，均归许可方或其供应商所有。可通过使用 eSeGeCe 组件访问的内容的所有权及知识产权均属于各自内容所有者，并可能受适用版权或其他知识产权法律及条约的保护。本最终用户许可协议不授予您使用此类内容的任何权利。所有未明确授予的权利均由许可方保留。

5. 不提供任何保证

许可方明确声明不对 eSeGeCe 组件提供任何担保。eSeGeCe 组件按“现状”提供，不附带任何形式的明示或暗示担保，包括但不限于对适销性、不侵权或特定用途适用性的担保。许可方不对 eSeGeCe 组件中包含的任何信息、文本、图形、链接或其他项目的准确性或完整性作出保证或承担责任。许可方对因传输计算机病毒、蠕虫、定时炸弹、逻辑炸弹或其他类似计算机程序而造成的任何损害不作任何担保。许可方进一步明确放弃对授权用户或任何第三方的任何担保或陈述。

6. 责任限制

在任何情况下，许可方均不对因“授权用户”使用或无法使用 eSeGeCe 组件而引起的任何损害（包括但不限于利润损失、业务中断或信息丢失）承担责任，即使许可方已被告知此类损害的可能性。在任何情况下，许可方均不对基于合同、侵权或其他原因造成的数据丢失、间接、特殊、附带、继发性（包括利润损失）或其他损害承担责任。许可方对

eSeGeCe 组件的内容或其任何部分（包括但不限于其中包含的错误或遗漏、诽谤、侵犯公众、隐私、商标权、业务中断、人身伤害以及隐私丢失、精神损害或机密信息披露）不承担任何责任。

索引

- AbstractApi 地理位置 1059
- Add Telegram Proxy 630
- ALPN 123
- Amazon SQS 910
- AMQP Channels 302
- AMQP QoS 313
- AMQP 事务 314
- AMQP 交换机 304
- AMQP 发布消息 309
- AMQP 消费消息 310
- AMQP 获取消息 312
- AMQP1 会话 336
- AMQP1 发送方链路 339
- AMQP1 发送消息 343
- AMQP1 接收方链路 341
- AMQP1 读取消息 345
- AMQP1 链路 338
- Anthropic 716
- Anthropic Files 740
- Anthropic MCP 连接器 742
- Anthropic Messages 721
- Anthropic Vision 723
- Anthropic Web Search 736
- Anthropic 工具使用 725
- Anthropic 引用 735
- Anthropic 扩展思考 729
- Anthropic 批处理 728
- Anthropic 提示缓存 733
- Anthropic 文档 731
- Anthropic 模型 727
- Anthropic 结构化输出 738
- API 404, 1058
- API 3Commas 532
- API Binance 406, 421, 426
- API Binance Futures 421, 426
- API Binance Futures Trade 426
- API Bitfinex 503
- API Bitget 582
- API Bitmex 495
- API Bybit 545
- API Cex Plus 567
- API Crypto.com 589
- API Deribit 586
- API Discord 571
- API Forex 549
- API GateIO 584
- API HTX 591
- API Kraken 449, 460, 467, 474, 482
- API Kraken Futures 467, 474, 482
- API Kraken Futures REST Private 482
- API Kraken REST Public 460
- API Kucoin 507
- API Kucoin Futures 519
- API MEXC 575
- API MEXC 期货, API 579
- API OKX 537
- API OpenAI 574
- API Pusher 487
- API SignalR, APIs 446
- API SignalRCore 440
- API Whatsapp 593
- API XTB 542
- API, API Blockchain 558
- API, API Coinbase, API Coinbase Pro 429
- API, API Kraken, API Kraken REST 私有 462
- API, Coinbase Pro 私有 REST API 436
- APIs 406, 421, 426, 433, 440, 449, 460, 467, 474, 482, 487, 495, 503, 532, 549, 555, 571, 575
- APIs, API Cex 560
- APIs, API Kraken, API Kraken Futures 465
- APIs, API Kraken, API Kraken Futures REST Public, API Kraken Futures 480
- APIs, API SocketIO 427
- APIs, API Telegram 615
- APIs, Binance 私有 REST API 416
- APIs, WebSockets, API Kraken, WebSockets Private 457
- APIs, WebSockets, API Kraken, WebSockets 公开频道 451
- APIs, WebSockets, Binance Connect 413
- APNs 815, 816

- Certificate-Based Connection [816](#)
- APNs, 通知请求, 通知请求: 发送, 发送, 发送: 通知请求 [814](#)
- APNs: 基于 Token 的连接 [815](#)
- Apple 推送通知 [812](#)
- Binance Subscribe [414](#)
- Binance 提币 [420](#)
- Binance 现货交易 [417](#)
- Binance 私有请求时间 [419](#)
- Binance 获取市场数据 [415](#)
- Bindings, 服务器, 服务器绑定 [171](#)
- Bitmex 订阅 WebSocket 频道 [500](#)
- Bitmex 连接 WebSocket API [499](#)
- Broadcast [112](#)
- Build [72](#)
- Build iOS Application [72](#)
- Certificate-Based Connection [816](#)
 - APNs [816](#)
- Channels [112](#), [302](#), [414](#)
- Client [142](#), [154](#), [163](#), [164](#), [189](#), [285](#), [301](#), [319](#), [391](#), [966](#), [974](#), [1062](#), [1063](#)
- Client AMQP Disconnect [301](#)
- Client AMQP1 Connect [329](#)
- Client AMQP1 身份验证 [333](#)
- Client AMQP1 连接状态 [332](#)
- Client Chat [1062](#)
- Client Kafka Disconnect [319](#)
- Client MQTT Connect [285](#)
- Client Open Connection [142](#)
- Client Register Protocol [164](#)
- Client Send Text [154](#)
- Client Send Text Message [154](#)
- Client WebSocket HandShake [163](#)
- Clients [319](#), [391](#), [1062](#)
 - Send Files [391](#)
- Coinbase Pro Place Orders [438](#)
- Coinbase Pro 沙盒账户 [439](#)
- Coinbase Pro 私有请求时间 [437](#)
- Coinbase Pro 获取市场数据 [435](#)
- Coinbase Pro 连接 [433](#)
- CommitOffset [324](#)
- Coturn [1082](#)
- Cross-Site Scripting [232](#)
- CryptoHopper [633](#)
- Datasnap [993](#)
- DeepSeek [747](#)
- DeepSeek Vision [750](#)
- DeepSeek 模型 [751](#)
- DeepSeek 消息 [749](#)
- Deflate-Frame [644](#)
- DNSBL [237](#)
- DPoP - 演示持有证明 (RFC 9449) [840](#)
- DPoP 验证 (RFC 9449) [871](#)
- E2EE 协议 [400](#)
- EPOLL [122](#)
- Error [200](#)
- FCrDNS [237](#)
- Files [114](#), [193](#), [391](#), [800](#)
- Flash [117](#)
- Flood Protection [235](#)
- Forex Connect [555](#)
- Forex Connect, REST API [551](#)
- Gemini [744](#)
- Gemini EmbedContent [760](#)
- Gemini Embeddings [760](#)
- Gemini Vision [756](#)
- Gemini 令牌计数, Gemini CountTokens [759](#)
- Gemini 函数调用 [761](#)
- Gemini 工具使用 [761](#)
- Gemini 模型 [757](#)
- Gemini 消息 [754](#)
- Gemini 结构化输出 [758](#)
- GetCommittedOffset [324](#)
- GetEarliestOffset [324](#)
- GetLatestOffset [324](#)
- Google Calendar [933](#)
- Google Calendar 服务账号 [941](#)
- Google Cloud FCM [942](#)
- Google Cloud Pub [925](#)
- Google 日历 [938](#), [939](#), [940](#)
- Google 日历 RefreshToken [940](#)
- Google 日历同步事件 [939](#)
- Google 日历同步日历 [938](#)
- Grok [769](#)
- Grok 模型 [773](#)
- Grok 消息 [771](#)
- Grok 视觉 [772](#)
- HeartBeat [108](#)
- HighPerfAcceptsPerWorker [211](#)
- HTML [851](#)
- HTTP [65](#), [111](#), [192](#), [193](#), [200](#), [201](#), [794](#), [798](#)
- HTTP Dispatch Files [193](#)

- HTTP 服务器流式视频 [203](#)
- HTTP, HTTPAPI, HTTPAPI 禁用 HTTP [215](#)
- HTTP, 转发 HTTP 请求 [124](#)
- HTTP/2 [194](#), [195](#), [197](#), [799](#), [800](#), [805](#), [808](#)
- HTTP/2 Download File [800](#)
- HTTP/2 Headers, HTTP/2 [802](#)
- HTTP/2 Server Push [195](#), [799](#)
- HTTP/2 断开连接原因 [805](#)
- HTTP/2 替代服务 [197](#)
- HTTP/2 服务器线程 [198](#)
- HTTP/2 部分响应, HTTP/2 [801](#)
- HTTP1 [818](#)
- HTTP2 [795](#)
- HTTPAPI FineTune [211](#)
- HTTPAPI URL 保留 [209](#)
- HTTPAPI 发送文件响应 [218](#)
- HTTPAPI 发送文本响应 [217](#)
- HTTPAPI, HTTPAPI 自定义请求头 [216](#)
- HTTPAPI, 触发, HTTPAPI OnDisconnect [220](#)
- ICE [980](#)
- ICE 收集候选者 [982](#)
- ICE 配对候选 [983](#)
- Indy [96](#)
- Indy SChannel [176](#), [204](#)
- IOCP [121](#)
- IoT [781](#), [789](#)
- IoT Azure MQTT Client [789](#)
- JWT [880](#)
- Kafka [319](#), [321](#), [324](#)
- Kafka Offsets [324](#)
- Kafka 消费消息 [321](#)
- Kafka, Kafka 主题, CreateTopic, DeleteTopic, GetMetadata [323](#)
- Kafka, Kafka 消费者组, 消费者组, GroupId, CommitSync [322](#)
- Kafka, Kafka 生产消息, 生产, 生产者 [320](#)
- KendoUI_Grid [1074](#)
- Kucoin Connect WebSocket API [512](#)
- Kucoin Futures Get Market Data [526](#)
- Kucoin Futures 私有 REST API [527](#)
- Kucoin Futures 订阅 WebSocket 频道 [525](#)
- Kucoin Futures 连接 WebSocket API [524](#)
- Kucoin Trade Spot [516](#)
- Kucoin 期货交易 [528](#)
- Kucoin 期货私有请求时间 [531](#)
- Kucoin 私有 REST API [515](#)
- Kucoin 私有请求时间 [518](#)
- Kucoin 获取市场数据 [514](#)
- Kucoin 订阅 WebSocket 频道 [513](#)
- Lightstreamer [555](#)
- LoadBalancing [135](#)
- MaxConnectionsPerIP [235](#)
- MaxMessagesPerSec [235](#)
- MCP [645](#)
- MCP Server Elicitation [660](#)
- MCP 客户端工具 [662](#)
- MCP 客户端引导 (Elicitation) [670](#)
- MCP 客户端提示 [664](#)
- MCP 客户端根目录 [668](#)
- MCP 客户端资源 [666](#)
- MCP 客户端采样 [669](#)
- MCP 服务器会话 [648](#)
- MCP 服务器工具 [649](#)
- MCP 服务器提示 [652](#)
- MCP 服务器根目录 [658](#)
- MCP 服务器资源 [655](#)
- MCP 服务器采样 [659](#)
- Message Filtering [232](#)
- Method [798](#)
- Mistral AI [774](#)
- Mistral Vision [778](#)
- Mistral 嵌入 [780](#)
- Mistral 模型 [779](#)
- Mistral 消息 [777](#)
- MQTT [285](#), [286](#), [287](#), [291](#), [292](#), [293](#), [294](#), [295](#), [789](#), [1064](#)
- MQTT Publish [294](#)
- MQTT Subscribe [291](#)
- MQTT 发布 [292](#)
- MQTT 发布消息 [292](#)
- MQTT 接收消息 [293](#)
- MQTT 清除保留消息 [295](#)
- MQTT, IoT, IoT Amazon MQTT 客户端 [782](#)
- MQTT, MQTT 主题 [290](#)
- MQTT, MQTT 发布订阅, MQTT 发布 [289](#)
- MQTT, 客户端 MQTT 版本, 客户端 [288](#)
- OAuth2 [808](#), [822](#), [851](#), [852](#), [853](#), [854](#)
- OAuth2 Register Apps [853](#)
- OAuth2 恢复访问令牌 [854](#)
- OAuth2 提供商 Azure AD [875](#)
- OAuth2 提供商私有端点 [876](#)
- OAuth2 提供商请求 [879](#)

- OAuth2 提供程序身份验证 [877](#)
- OAuth2 自定义登录 [851](#)
- OAuth2, Google OAuth2 密钥 [915](#)
- OAuth2, OAuth2 无身份验证 URL [856](#)
- OAuth2, 服务器, 服务器示例 [847](#)
- OAuth2, 服务器, 服务器认证 [855](#)
- Ollama [763](#)
- Ollama 嵌入 [768](#)
- Ollama 模型 [767](#)
- Ollama 消息 [766](#)
- ompClassic [211](#)
- ompHighPerf [211](#)
- OpenAI [671](#)
- OpenAI Audio [682](#), [699](#)
- OpenAI Completion [679](#)
- OpenAI Edit [681](#)
- OpenAI Fine-Tuning [687](#)
- OpenAI RealTime [684](#)
- OpenAI 上传 [689](#)
- OpenAI 内容审核 [683](#)
- OpenAI 响应 [685](#)
- OpenAI 应用程序 [690](#)
- OpenAI 批处理 [688](#)
- OpenAI 聊天 [680](#)
- OpenAI 语音 [686](#)
- OpenAPI [998](#)
- OpenAPI Amazon AWS [1019](#)
- OpenAPI Amazon AWS S3 [1027](#)
- OpenAPI Amazon AWS 凭证 [1025](#)
- OpenAPI API [1011](#)
- OpenAPI Google Cloud [1028](#)
- OpenAPI Google Cloud Calendar [1041](#)
- OpenAPI Google Cloud OAuth2 [1033](#)
- OpenAPI Google Cloud PubSub [1040](#)
- OpenAPI Google Cloud 服务账号 [1036](#)
- OpenAPI Google Drive [1042](#)
- OpenAPI Microsoft [1044](#)
- OpenAPI Microsoft Graph [1057](#)
- OpenAPI Microsoft OAuth2 代码 [1053](#)
- OpenAPI Microsoft OAuth2 凭据 [1055](#)
- OpenAPI Microsoft Tenant [1049](#)
- OpenAPI Microsoft 注册应用程序 [1050](#)
- OpenAPI Server [1015](#)
- OpenAPI 命令行 [1007](#)
- OpenAPI 客户端 [1012](#)
- OpenAPI 库 [1009](#)
- OpenAPI 服务器代码优先 [1018](#)
- OpenAPI 服务器规范优先 [1017](#)
- OpenAPI 解析器 Pascal [1000](#)
- OpenAPI 附加属性 [1005](#)
- OpenSSL [80](#), [87](#), [88](#)
- OpenSSL Android [87](#)
- OpenSSL iOS [88](#)
- OpenSSL P12 证书 [92](#)
- OpenSSL 加载附加函数 [94](#)
- OpenSSL 自有 CA 证书 [90](#)
- OpenSSL 验证证书 [93](#)
- OpenSSL, OpenSSL OSX [85](#)
- OpenSSL, OpenSSL Windows [83](#)
- OpenSSL, OpenSSL 证书 [151](#)
- OperatingMode [211](#)
- PDF-Back-Cover [1094](#)
- PDF-Front-Cover [1](#)
- PerMessage-Deflate [643](#)
- Pinecone [713](#)
- Poll [321](#)
- Post Big Files [114](#)
- Protocol AMQP1 [325](#)
- Protocol Dataset [381](#)
- Protocol Dataset Javascript [381](#)
- Protocol Default [373](#)
- Protocol Default Javascript [373](#)
- Protocol Files [386](#)
- Protocol MQTT [283](#)
- Protocol WAMP [358](#)
- Protocol WAMP Javascript [358](#)
- Protocol WAMP2 [366](#)
- Protocol WebRTC [352](#), [354](#)
- Protocol WebRTC Javascript [354](#)
- Protocols [164](#), [283](#), [352](#), [354](#), [358](#), [366](#), [373](#), [381](#), [386](#)
- Proxy [137](#), [165](#), [630](#)
- QueueLength [211](#)
- QuickStart HTTP [65](#)
- Rate Limiting [235](#)
- RCON [632](#)
- Request HTTP [798](#)
- Resource Owner Password Credentials Grant (RFC 6749) [836](#)
- Response Body [200](#)
- RTCMultiConnection [637](#)
- RTCPeerConnection DTLS [991](#)

RTCPeerConnection ICE 990
 RTCPeerConnection STUN TURN 988
 RTCPeerConnection WebSocket 客户端 987
 RTCPeerConnection WebSocket 服务器 986
 RTCPeerConnection 信令 989
 RTCPeerConnection 数据 992
 SChannel 176, 204
 SChannel Server 176, 204
 SChannel 获取连接信息 153
 SChannel 证书 152
 Send Files 391
 Clients 391
 Send Files To Clients 391
 Server 170, 189, 194, 201, 799, 852
 Server Endpoints 852
 Server Read Headers 189
 Server Sessions 201
 Server SSL SChannel 176, 204
 Server Start 170
 Server, Server SSL, HTTPAPI 213
 ServerSentEvents 1076
 Service 197
 Service, Google 服务账号 921
 SkipIOCPOnSuccess 211
 SQL Injection 232
 STUN 961, 965, 966, 968
 STUN Client Attributes 966
 STUN 客户端长期凭据 965
 STUN 服务器长期凭证 968
 Sub 925
 SubProtocol 131
 TCP 连接 130
 Telegram 聊天 626
 Telegram 获取超级群组成员 629
 Telegram 赞助消息 627
 Throttle 132
 TsgcAIChat 705
 TsgcAIDatabaseVectorFile 709
 TsgcAIDatabaseVectorPinecone 710
 TsgcAIOpenAIAssistant 691
 TsgcAIOpenAIAssistant 函数调用 697
 TsgcAIOpenAIAssistant 文件搜索 693
 TsgcAIOpenAIAssistant 流式传输 695
 TsgcAIOpenAIChatBot 706
 TsgcAIOpenAIEmbeddings 708
 TsgcAIOpenAITranslator 707
 TsgcAudioPlayerMCI 701
 TsgcAudioRecorderMCI 700
 TsgcBotClassification 237
 TsgcHTTP_API_Anthropic 720
 TsgcHTTP_API_DeepSeek 748
 TsgcHTTP_API_Gemini 746
 TsgcHTTP_API_Grok 770
 TsgcHTTP_API_Mistral 776
 TsgcHTTP_API_Ollama 765
 TsgcHTTP_API_OpenAI 678
 TsgcHTTP_JWT_Client 882
 TsgcHTTP_JWT_Server 885
 TsgcHTTP_OAuth2_Client 823
 TsgcHTTP_OAuth2_Client_Google 829
 TsgcHTTP_OAuth2_Client_Microsoft 830
 TsgcHTTP_OAuth2_Server 844
 TsgcHTTP_OAuth2_Server_Provider 873
 TsgcHTTP2Client 796
 TsgcHTTP2ConnectionClient 809
 TsgcHTTP2RequestProperty 810
 TsgcHTTP2ResponseProperty 811
 TsgcICEClient 981
 TsgcIWebsocketClient 274
 TsgcIWWSPClient_Dataset 380
 TsgcIWWSPClient_sgc 372
 TsgcRTCPeerConnection 984
 TsgcServerHTTPAPI_FineTune 211
 TsgcSTUNClient 962
 TsgcSTUNServer 967
 TsgcTCPClient 166
 TsgcTextToSpeechAmazon 704
 TsgcTextToSpeechGoogle 703
 TsgcTextToSpeechSystem 702
 TsgcTURNClient 971
 TsgcTURNServer 977
 TsgcUDPClient 959
 TsgcUDPServer 960
 TsgcWebPush_Client 641
 TsgcWebsocketClient 139
 TsgcWebsocketClient_WinHTTP 221
 TsgcWebsocketFirewall 222, 232, 235, 237
 TsgcWebsocketHTTPServer 190
 TsgcWebsocketHTTPServer_Sessions 201
 TsgcWebsocketLoadBalancerServer 272
 TsgcWebsocketProxyServer 273
 TsgcWebsocketServer 168

- TsgcWebSocketServer_HTTPAPI 207
- TsgcWebView2 945
- TsgcWSAPIClient_MCP 661
- TsgcWSAPIKeyManager 261
- TsgcWSAPIServer_MCP 646
- TsgcWSAPIServer_WebAuthn 887
- TsgcWSAPIServer_WebPush 640
- TsgcWSCircuitBreaker 251
- TsgcWSConnection 276
- TsgcWSHTTP2WebBrokerBridgeServer 996
- TsgcWSHTTPWebBrokerBridgeServer 994
- TsgcWSMessageFile 389
- TsgcWSPClient_AMQP 297
- TsgcWSPClient_AMQP1 326
- TsgcWSPClient_Dataset 379
- TsgcWSPClient_E2EE 403
- TsgcWSPClient_Files 388
- TsgcWSPClient_Kafka 316
- TsgcWSPClient_MQTT 284
- TsgcWSPClient_Presence 396
- TsgcWSPClient_sgc 371
- TsgcWSPClient_STOMP 347
- TsgcWSPClient_STOMP_ActiveMQ 349
- TsgcWSPClient_STOMP_RabbitMQ 348
- TsgcWSPClient_WAMP 357
- TsgcWSPClient_WAMP2 367
- TsgcWSPPresenceMessage 395
- TsgcWSPServer_AppRTC 351
- TsgcWSPServer_Dataset 378
- TsgcWSPServer_E2EE 402
- TsgcWSPServer_Files 387
- TsgcWSPServer_Presence 394
- TsgcWSPServer_sgc 370
- TsgcWSPServer_WAMP 356
- TsgcWSPServer_WebRTC 353
- TsgcWSRateLimiter 240
- TsgcWSServer_HTTPAPI_WebBrokerBridge 997
- TURN 970, 974, 978
- TURN Client Create Permissions 974
- TURN 服务器长期凭证 978
- Wait Response 294
- WAMP 364
- WAMP RPC Progress Results 364
- WAMP, WAMP 发布者 362
- WAMP, WAMP 简单 RPC 363
- WAMP, WAMP 订阅者 361
- WatchDog 109
- Web 浏览器测试 102
- WebAuthn 886
- Webauthn Javascript 客户端 906
- WebAuthn MDS 902
- WebAuthn 授权 903
- WebAuthn 授权 HTTP 904
- WebAuthn 授权 WebSocket 905
- WebAuthn 注册 890
- WebAuthn 注册响应 894
- WebAuthn 注册结果 896
- WebAuthn 注册请求 893
- WebAuthn 认证结果 901
- WebAuthn 身份验证 897
- WebAuthn 身份验证响应 900
- WebAuthn 身份验证请求 899
- WebPush 639
- WebRTC, 服务器 1077
- WebSocket Events 98
- WebSocket 连接参数, WebSocket 99
- WebSocket 重定向 149
- WebSockets 98, 149, 163, 414, 433, 467, 474, 555
- WebSockets Private 474
- WebSockets Public 467
- WebSockets, WebSockets 快速入门 63
- WebSockets, 连接 WebSocket 服务器 141
- WebSockets, 通道, Coinbase Pro 订阅 434
- WebSockets, 频道, 外汇订阅, Lightstreamer 556
- WebView2 Cookie 951
- WebView2 JavaScript 949
- WebView2 下载 952
- WebView2 导航 947
- WebView2 设置 954
- WebView2 高级功能 956
- WhatsApp Token 598
- WhatsApp Webhook 599
- WhatsApp 下载媒体 614
- WhatsApp 创建应用程序 596
- WhatsApp 发送交互式消息 604
- WhatsApp 发送文件 612
- WhatsApp 发送模板消息 608
- WhatsApp 发送消息 601
- WhatsApp 安全 600

- WhatsApp 手机号码 ID [597](#)
- WhatsApp 接收消息和状态通知 [610](#)
- Windows 服务 [101](#)
- XSS [232](#)
- 上传文件 [1071](#)
- 丢失的断开连接 [146](#)
- 事务 [129](#)
- 二进制消息 [156](#)
- 令牌内省 (RFC 7662) [869](#)
- 令牌撤销 (RFC 7009) [867](#)
- 价格刻度 [554](#)
- 价格柱 [554](#)
- 使用 DLL [100](#)
- 保持客户端连接活跃, 客户端, 客户端列表 [804](#)
- 内存管理器 [77](#)
- 分片消息 [138](#)
- 刷新令牌授权 [863](#)
- 协议 [278](#), [346](#), [369](#)
- 协议 STOMP [346](#)
- 协议, AMQP 协议 [296](#)
- 协议, Kafka 协议 [315](#)
- 协议, Presence 协议, Presence 协议 JavaScript [397](#)
- 协议, Protocol AppRTC [350](#)
- 协议, Protocol Presence [393](#)
- 协议, WAMP 协议 [355](#)
- 协议, 协议 Javascript [280](#)
- 协议, 自定义子协议 [103](#)
- 协议数据集, 协议 [377](#)
- 协议数据集, 协议, 协议数据集复制表 [384](#)
- 协议数据集, 协议, 协议数据集通知更新 [385](#)
- 压缩 [116](#)
- 反向 DNS [237](#)
- 发送 Telegram 发票消息 [628](#)
- 发送 Telegram 粗体文字消息 [625](#)
- 发送带内联按钮的 Telegram 消息 [623](#)
- 发送带按钮的 Telegram 消息 [623](#), [624](#)
- 基于 Token 的连接 [815](#)
- 基于 Token 的连接: APNs [815](#)
- 外汇交易, 订单, 止损限价 [552](#)
- 外汇行情数据 [554](#)
- 外汇账户, 持仓, 订单, 历史 [553](#)
- 如何下 Bitmex 订单 [501](#)
- 安全连接 [106](#)
- 安装 [29](#)
- 安装 sgclndy 包 [55](#)
- 安装设置 [32](#)
- 安装软件包 [43](#)
- 安装错误 [50](#)
- 客户端 [144](#), [156](#), [162](#), [165](#), [287](#), [300](#), [803](#), [807](#), [965](#), [1064](#), [1064](#), [1070](#)
- 客户端 AMQP 连接 [300](#)
- 客户端 AMQP1 Azure MessageBus [334](#)
- 客户端 AMQP1 断开连接 [330](#)
- 客户端 AMQP1 空闲超时连接 [331](#)
- 客户端 MQTT 会话 [287](#)
- 客户端, STUN, STUN 客户端 UDP 重传 [964](#)
- 客户端, Telegram 客户端 [1080](#)
- 客户端, TURN 客户端分配 IP 地址, TURN [973](#)
- 客户端, TURN 客户端发送指示, TURN [975](#)
- 客户端, 客户端 SocketIO [1066](#)
- 客户端, 客户端, 客户端 Kafka Connect, Kafka [318](#)
- 客户端, 客户端保持连接 [145](#)
- 客户端代理 [165](#)
- 客户端们 [287](#), [300](#)
- 客户端关闭连接 [144](#), [803](#)
- 客户端凭据授权 [859](#)
- 客户端凭据授权 (RFC 6749) [835](#)
- 客户端发送二进制消息, 客户端 [155](#)
- 客户端发送文本 [156](#)
- 客户端异常 [162](#)
- 客户端待处理请求, 客户端, 客户端列表 [806](#)
- 客户端快照 [1070](#)
- 客户端身份验证 [807](#)
- 客户端连接 [183](#)
- 嵌入创建向量 [711](#)
- 嵌入向量聊天机器人 [712](#)
- 已找到 [626](#)
- 带 PKCE 的授权码流程 (RFC 7636) [833](#)
- 扩展 [642](#)
- 授权码授权 (RFC 6749) [831](#)
- 授权码许可 [857](#)
- 接收二进制消息 [158](#)
- 接收二进制消息, 服务器 [188](#)
- 接收文本消息 [157](#), [187](#)
- 数据中心检测 [237](#)
- 文件 [136](#), [1071](#)
- 文件, 发送大文件 [392](#)

文件, 服务器, 服务器: 发送文件, 向服务器发送文件, 发送文件, 发送文件: 服务器 390
日志 110
暴力破解, 自动封禁, IP 封禁, TsgcWebSocketFirewall 230
服务器 185, 186, 187, 192, 286, 968, 978, 1060, 1073
服务器, STUN, STUN 服务器备用服务器 969
服务器, TURN, TURN 服务器分配 979
服务器, 服务器 AppRTC 1078
服务器, 服务器 SSL 174
服务器, 服务器保持活跃 173
服务器, 服务器保持连接活跃 180
服务器, 服务器关闭连接 182
服务器, 服务器启动关闭 172
服务器, 服务器快照 1069
服务器, 服务器普通 TCP 181
服务器, 服务器监控 1067
服务器发送事件 133
服务器发送二进制消息 186
服务器发送文本消息 185
服务器聊天 1060
服务器请求 192
服务器身份验证 1073
服务器验证证书 179
服务质量, 质量, 质量: 服务, 服务, 服务: 质量 125
机器人 626
机器人检测 237
构建 68, 71
构建 Android 应用程序 71
构建, 构建 OSX 应用程序 69
桌面应用程序的 OAuth2 客户端 828
概述 61
注册, 注册 Telegram 用户 631

流程, 流程: 线程, 线程, 线程: 流程 67
消费 321
消费者 321
爬虫检测 237
版本 27
生成 813
生成: 远程通知 APNs 813
简介 23
绑定 113
群组 119
自定义对象 118
自签名证书 214
订阅 321
认证, 服务器, 服务器认证 184
许可证 1084
设备授权许可 (RFC 8628) 838, 865
资源所有者密码凭证授权 861
身份验证 104
身份验证, 客户端身份验证, 客户端 159
远程通知 APNs 813
远程通知 APNs: 生成 813
连接 Mosquitto 286
连接 TCP 服务器 147
连接 TIME_WAIT 148
连接安全服务器 150
适用于 Web 应用程序的 OAuth2 客户端 827
配置 ZLib 60
配置安装 53
队列 127
队列, AMQP 队列 306
防火墙黑名单, 防火墙白名单, IP 过滤, CIDR, TsgcWebSocketFirewall 228
频道, 客户端, TURN 976
高性能服务器 74
默认协议 369

